UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Uma Biblioteca de Tipos Abstratos de Dados para a Área de Análise e Projeto de Sistemas de Controle.

Dissertação submetida à Universidade Federal de Santa Catarina
para a obtenção do grau de
Mestre em Engenharia Elétrica

Márcio Heidi Suguieda

Florianópolis, Setembro de 1993.

Uma Biblioteca de Tipos Abstratos de Dados para a Área de Análise e Projeto de Sistemas de Controle.

Márcio Heidi Suguieda

Esta dissertação foi julgada para obtenção do título de
Mestre em Engenharia
especialidade **Engenharia Elétrica,**
área de concentração **Sistemas de Controle e Automação Industrial,**
e aprovada em sua forma final pelo Curso de Pós-Graduação.

Prof. Edson Roberto De Pieri, Dr.
Orientador

Prof. Roberto de Souza Salgado, Ph.D.
Coordenador do Curso de Pós-Graduação em Engenharia Elétrica

Banca Examinadora:

Prof. Edson Roberto De Pieri, Dr.
Orientador

Prof. Julio Elias Normey Rico, M.Sc.
Co-orientador

Prof. Eugênio de Bona Castelan Neto, Dr.

Prof. Ivanil Sebastião Bonatti, Dr.

À minha família e a todos
que estimam o verdadeiro
significado da palavra amizade.

MENSAGEM

"Continuarei a trilhar meu caminho, almejando mais uma etapa do viver. Caminharei com alento, sabendo que fiz grandes amigos e que, a cada instante findo, compreendo melhor o significado do vocábulo 'família'. Prosseguirei enfim, aprendendo o viver ... simplesmente vivendo ..."

O autor.

AGRADECIMENTOS

Sinceros agradecimentos ao Prof. Dr. Edson Roberto De Pieri que demonstrou ser, antes de um ótimo orientador, um grande amigo. Igualmente pela amizade e pela co-orientação, agradecimentos ao Prof. M.Sc. Julio Elias Normey Rico. Um muito obrigado também ao Prof. Dr. José Eduardo Ribeiro Cury pela sua atenção e orientação durante a fase de créditos do programa de pós-graduação.

Agradecimentos aos membros da banca examinadora pela aceitação de participação e pelas críticas e comentários.

Agradecimentos aos professores, funcionários e colegas de departamento que, direta ou indiretamente, contribuíram na realização deste projeto.

Pela ajuda na solução de grandes problemas, humildes agradecimentos ao Prof. Dr. Eugênio de Bona Castelan Neto, aos analistas Jorge Hermógenes de Souza (LCMI), Roberto Becker Rostirolla (PGEEL), José Antônio Alves Duarte (NPD) e aos mestres em engenharia Leonardo César Kammer, Saul Silva Caetano e Paulo Roberto de Oliveira Valim.

Aos companheiros na jornada: Maurício, Marcos, Elieser, Iberê, Assis, Juan, Roberto, Sérgio, Romeu, Vinicius, Arão, Bazanella, Tristão, Zorzo, Eliane, Keiko, Nacamura, Thereza, Carla, Luís Otávio, Idmilson, Busch, João Manoel, Adriana, Claudiane, Bianca, Claudine, Joable, Aldebaro, Polyana, Jáder, Cícero, Márcio Có, Falcondes, Aberides, Fernando Mussoi, Denise, Yolando, Damasceno, Gisele, Fernanda, Eduardo e Carlos Sica, deixo aquele "Valeu turma! Felicidades para todos!". E para as engenheiras Denise Maria Maldonado da Cunha, Michella de Mendonça Dantas e Lilian Vasconcellos França fica aquele sempre obrigado pela troca de idéias e de ideais.

Fica também uma menção de agradecimento à UFSC e à CAPES pelo suporte material e financeiro.

Por fim, um simples e eterno obrigado aos meus familiares que, apesar da distância, sempre souberam como transmitir carinho e apoio à minha vida.

A todos que ajudaram ...

Um Muito Obrigado!...

SUMÁRIO

ÍNDICE DE FIGURAS

ÍNDICE DE QUADROS

SIGLAS E NOTAÇÕES

ASCII	*The American Standard Code for Information Interchange (também conhecida como USASCII - The USA Standard Code for Information Interterchange)*
CACE	*Computer Aided Control Engineering*
CPGEEL	Curso de Pós-Graduação em Engenharia Elétrica
CTC	Centro Tecnológico
DOS	*Disk Operating System*
EEL	Departamento de Engenharia Elétrica
FTP	*File Transfer Protocol*
IBM®	*Industrial Business Machine, Inc.*
IEEE	*The Institute of Electrical and Electronics Engineers, Inc.*
LCMI	Laboratório de Controle e Microinformática
LOO	Linguagem Orientada a Objetos
NPD	Núcleo de Processamento de Dados
PACSC	Projeto Assistido por Computador para a área de Sistemas de Controle
PC	*Personal Computer*
PGEEL	Pós-Graduação em Engenharia Elétrica
POO	Programação Orientada a Objetos
UFSC	Universidade Federal de Santa Catarina

RESUMO

Este trabalho é o resultado da elaboração de um projeto de base visando "**Uma Biblioteca de Tipos Abstratos de Dados para a Área de Análise e Projeto de Sistemas de Controle**". Esta biblioteca foi composta por nove módulos estruturados, destinadas a facilitar a representação e a manipulação de vetores, matrizes, polinômios, funções de transferência e sistemas de equações de estado. De fato, este projeto marca o início da biblioteca pois, além do material já implementado, foi colocado à disposição toda uma metodologia para a continuidade dessa linha de trabalho.

Esta metodologia deu ênfase, primordialmente, aos temas reutilização e manutenção de *software*, aplicados ao desenvolvimento de recursos voltados aos sistemas do tipo *CACE* (*Computer Aided Control Engineering*). Aliás, o projeto envolveu uma integração entre a Engenharia de Controle e a Engenharia de *Software*.

Foram ainda escolhidas, para atingir o objetivo, as facilidades oferecidas pela linguagem de programação C++, visando o paradigma de programação orientada a objetos, e pelas estações de trabalho Sun® SPARC[1], operando sob o sistema UNIX®[2].

---

[1] Sun e SPARC são marcas registradas de Sun Microsystems, Inc.
[2] UNIX é marca registrada de AT&T.

ABSTRACT

*This work is the result from an elaboration of a support project aiming at "A **Library of Abstract Data Types for the Area of Analysis and Design of Control Systems**". This library has been arranged by nine structured modules, directed to make easier the representation and manipulation of vectors, matrices, polynomials, transfer functions and state variables systems. Actually, this project seals the beginning of the library since, besides the material already implemented, it has been put ready for use a methodology for the continuity of this line of work.*

This methodology emphasizes, primordialy, the topics software reutilization and maintenance, applied to the development of resources faced to CACE (Computer Aided Control Engineering) systems. By the way, the project has involved the integration between Control Engineering and Software Engineering.

And also, to reach purpose, it has been chosen the facilities offered by the C++ programming language, driving at the object-oriented programming paradigm, and by the Sun® SPARC[1] workstations, operating under UNIX®[2] system.

---

[1] *Sun e SPARC are registered trademarks of Sun Microsystems, Inc.*
[2] *UNIX is a registered trademark of AT&T.*

CAPÍTULO 1

INTRODUÇÃO

No âmbito mundial, o desenvolvimento de ferramentas de análise e projeto à disposição do engenheiro de controle vem sofrendo, nestas últimas quatro décadas, uma evolução admirável na busca do estado da arte nesta área [Astrom 85, Fontanini 90]. Praticamente, esta evolução ocorreu do lápis e papel, ou da programação em computadores analógicos, aos atuais ambientes de programação capazes de modelar, identificar, analisar, simular e projetar sistemas de controle [Astrom 83]. Ambientes estes que já contam inclusive com a capacidade de usufruto de conhecimentos heurísticos ou empíricos - os chamados sistemas baseados em conhecimento.

Todavia, este desenvolvimento nem sempre caminhou de forma linear e organizada, de modo que problemas associados com estruturação, portabilidade, confiabilidade, eficiência, integração homem-máquina, e principalmente manutenção, passaram a afligir cada vez mais os projetistas de *software*. Para tornar ainda a situação menos favorável, aparece também a constatação de que a produção de *software* não consegue evoluir à mesma taxa que o desenvolvimento de *hardware*, propiciando assim indícios de que tal fator tende a ser um dos gargalos do progresso da área de controle de processos [Astrom 85].

Portanto, à medida que estas preocupações concretizam-se, novos conceitos de desenvolvimento de sistemas computacionais surgem com a Engenharia de *Software*, que integrada à Engenharia de Controle, possibilita a busca de meios mais adequados de otimizar a produção e a qualidade de ferramentas de análise e projeto à disposição do engenheiro de controle.

1.1. - Engenharia de Controle x Engenharia de *Software*

Essa integração entre a Engenharia de Controle e a Engenharia de *Software* possibilita, atualmente, um desenvolvimento melhor sustentável dos sistemas computacionais

destinados à área de controle de processos, já que a primeira fornece as metodologias de resolução dos problemas e a segunda propicia as sistematologias de produção e de manutenção dos *softwares*.

No Brasil, a necessidade dessa integração é marcante, porquanto a produção brasileira de ferramentas de auxílio ao engenheiro de controle está apenas criando a sua base [Farines 86] e, conseqüentemente, não atinge ainda destaque a nível mundial. Por outro lado, esta consiste em uma área que merece investimento, pois além de ser reconhecidamente consolidada no mundo [Ferreira 87, Fontanini 90], não exige, a princípio, recursos humanos ou financeiros que estejam fora da realidade brasileira, apresentando assim condições favoráveis de evolução científica e tecnológica.

Além disso, ferramentas de auxílio ao engenheiro podem certamente promover um aumento da produtividade do profissional, assim como incentivá-lo ao estudo e aplicação de novas técnicas de análise e projeto de sistemas de controle [Spang III 85, Ferreira 87, Fontanini 90]. E a propósito, hoje as ferramentas de auxílio ao engenheiro já podem ser consideradas indispensáveis para os profissionais envolvidos com a área de controle de processos.

1.2. - O Projeto

Uma das linhas de trabalho do Laboratório de Controle e Microinformática do Departamento de Engenharia Elétrica da Universidade Federal de Santa Catarina (LCMI - EEL - UFSC) é voltada à área de Projeto Assistido por Computador para a área de Sistemas de Controle (PACSC) e visa justamente atender as necessidades afins das instituições de ensino, pesquisa e desenvolvimento tecnológico, assim como do meio industrial. A produção nacional de trabalhos nessa linha, como mencionado anteriormente, ainda é pouco expressiva, devido a motivos que vão desde a situação político-econômica, passando pelo nível de produção estrangeira até a simples dificuldade de reutilização e manutenção dos pacotes computacionais. Aliás, essas razões foram e são mais do que suficientes para que muitos projetos brasileiros, principalmente no ambiente acadêmico, deixassem de ser devidamente efetivados.

Atentos a esta realidade, foi decidido principiar, neste projeto, o planejamento e implementação de uma biblioteca básica para a área de controle de processos, contendo módulos que possam servir de base para outros projetos e, em particular, para os relativos ao desenvolvimento de sistemas do tipo *CACE* (*Computer Aided Control Engineering*). Estas

bibliotecas devem ser descritas por estruturas que facilitem o uso de vetores, matrizes, polinômios, funções de transferência e sistemas de equações de estado, quando da elaboração, a príncipio em estações de trabalho, de programas computacionais na linguagem C++.

1.3. - Divisão do Trabalho

O texto do trabalho foi elaborado procurando sempre respeitar uma continuidade de assunto, assim como fornecer todas as explicações necessárias para as decisões de projeto tomadas. Sucintamente, este texto expõe, no capítulo 2, uma descrição do projeto, onde são detalhados a motivação e o propósito do mesmo, as especificações preliminares estabelecidas, as justificativas de desenvolvimento, tanto para a linguagem C++ como para as estações de trabalho Sun® SPARC em conjunção com o sistema operacional UNIX® e, também, um cronograma descrevendo as etapas do trabalho.

O capítulo 3 é relativo à implementação e expõe, por sua vez, o objeto de implementação do projeto, sob o aspecto da programação orientada a objetos e da linguagem C++. Além disso, são acrescentadas especificações concernentes aos critérios adotados na programação, às convenções de projeto e aos padrões sugeridos ao laboratório, relativos ao desenvolvimento e à documentação de material produzido na linguagem C++. Neste capítulo ainda, há uma seção especialmente destinada a situar o trabalho sob o enfoque dos sistemas *CACE*. Convém entretanto salientar que, em alguns pontos deste capítulo, um prévio conhecimento da linguagem C++ pode facilitar a compreensão do texto, porquanto estes apresentam uma abordagem técnica voltada à codificação propriamente dita.

Já o capítulo 4 coloca um resumo dos resultados de projeto, basicamente na forma de quadros, elucidando assim o produto obtido: a biblioteca de tipos abstratos de dados destinados a área de análise e projeto de sistemas de controle. Alguns resultados numéricos também são apresentados para exemplificar o potencial dos métodos implementados.

As conclusões e perspectivas deste trabalho podem ser devidamente encontradas no capítulo 5. Após este capítulo, estão disponíveis o conjunto de referências bibliográficas e três anexos contendo, nessa ordem, uma cópia da norma interna sugerida ao LCMI para o desenvolvimento e documentação de classes C++ [Suguieda 93a], um exemplo de documentação impressa elaborada para as classes C++ e um exemplo de documentação *on-line*

(*man pages*) para as estações de trabalho. Cabe ainda salientar que a estrutura de cada capítulo apresenta, além de uma introdução, conclusões e referências específicas a cada um deles.

1.4. - Referências Específicas

[Aström 83] ÅSTRÖM, Karl Johan. Computer Aided Modeling, Analysis and Design of Control Systems - A Perspective. *IEEE Control System Magazine*, v. 3, p. 4 - 16, maio 1983.

[Aström 85] ÅSTRÖM, Karl Johan. Process Control - Past, Present and Future. *IEEE Control Systems Magazine*, v. 5, p. 3 - 10, ago. 1985.

[Farines 86] FARINES, Jean-Marie, SAVI, Vanio M., BRUCIAPAGLIA, Augusto H. Projeto Assistido por Computador para Sistemas de Controle: Um Pacote Interativo. *Anais do 6º. CBA - Congresso Brasileiro de Automática*, UFMG, Belo Horizonte - M.G., v. 1, p. 550 - 554, nov. 1986.

[Ferreira 87] FERREIRA, P. A. V., FONTANINI, W., GUERRA, A. C., AMARAL, W. C., GOMIDE, F. A. C., Modelagem, Análise e Projeto de Sistemas de Dinâmicos Integrados por Computador. *Revista SBA: Controle & Automação*, v. 1, n. 4, p. 322 - 330, out. 1987.

[Fontanini 90] FONTANINI, W., SILVA Fº., O. S., FERREIRA, P. A. V. Um Ambiente Integrado para Análise e Projeto no Espaço de Estados. *Anais do 8º. CBA - Congresso Brasileiro de Automática*, UFPa, Belém - P.A., v. 1, p. 189 - 194, 1990.

[Spang III 85] SPANG III, H. Austin. Experience and Future Needs in Computer-Aided Control Design. *IEEE Control Systems Magazine*, v. 5, p. 18 - 21, fev. 1985.

[Suguieda 93a] SUGUIEDA, Márcio Heidi, KAMMER, Leonardo César. Norma para Documentação de Classes C++. *Publicação Interna LCMI PI 93-1*, LCMI - UFSC, Florianópolis - S.C., jun. 1993.

CAPÍTULO 2

DESCRIÇÃO DO PROJETO

2.1. - Introdução

Planejar e implementar ferramentas de auxílio ao engenheiro de controle não são tarefas simples de serem realizadas, principalmente quando se pretende obter um produto de qualidade. Em verdade, estas tarefas são bastantes complexas [Savi 88] e normalmente de longa duração. Num ambiente acadêmico em particular, todo projeto que essencialmente possui alguma etapa de implementação merece uma atenção especial, porquanto a realidade atual brasileira assim o exige. Ou seja, medidas cautelosas devem ser tomadas no intuito de precaver problemas que impossibilitem a realização do projeto e que, por conseguinte, sejam capaz de ponderar aspectos como a disponibilidade presente e futura de equipamentos, a limitação de tempo hábil, a quantificação e qualificação dos recursos humanos, materiais e financeiros, e até mesmo a política educacional do país.

Enfatiza-se ainda que as universidades apresentam alta rotatividade de recursos humanos, particularmente em termos de programas de mestrado, exigindo assim uma constante formação destes recursos, a qual aliás, caracteriza o objetivo maior de uma instituição de ensino. Isto também contribui em muito para que trabalhos de longa duração, neste âmbito, necessitem de uma combinação renovada de esforços ao longo do tempo e de uma separação bem definida das etapas a serem realizadas. De fato, todos estes aspectos são do cotidiano daqueles que pertencem ao meio acadêmico, e por isso, perdem um pouco do seu fascínio. Todavia, ter consciência destes aspectos, certamente pode contribuir para uma melhor evolução de um projeto do tipo.

Considerando toda essa dificuldade de planejamento e desenvolvimento de projeto envolvendo implementação de *software*, foram determinados neste capítulo as metas gerais deste projeto de mestrado sob uma expectativa mais concreta e realística, assim como relatados as motivações que levaram à efetivação deste (seção 2.2.). Na seção 2.3. são descritos as especificações do trabalho, na seção 2.4. as justificativas quanto às decisões de projeto e ainda

na 2.5. são estabelecidas as etapas de projeto. A seção 2.6. apresenta as conclusões relativas à descrição do projeto e, finalmente, podem ser encontradas na última seção as referências bibliográficas específicas a este capítulo.

2.2. - Motivação e Propósitos Gerais do Projeto

O desenvolvimento e a utilização de ambientes computacionais oriundos da área *CACE* (*Computer Aided Control Engineering*) têm crescido em importância de forma acentuada ao longo desta última década. Durante este período, esta área consolidou-se como linha de pesquisa com características próprias, a ponto de ser reconhecida recentemente como uma das duas principais áreas de pesquisa em controle automático, sendo a outra relativa à atividade básica de desenvolvimento de novas metodologias e aplicações na Teoria de Controle. Motivações para o desenvolvimento nesta linha têm origem na necessidade de se promover um aumento de produtividade do engenheiro de controle em tarefas de modelagem, análise e projeto face ao crescente volume e complexidade de novos sistemas e, ao mesmo tempo situar o ensino e a pesquisa da Teoria de Controle dentro de um enfoque moderno e integrador [Ferreira 87, Fontanini 90].

O trabalho desenvolvido no âmbito desta dissertação enquadra-se num projeto de maior complexidade, cujo objetivo é realizar um simulador para análise e controle de processos industriais, a partir de uma biblioteca de funções básicas e com uma concepção modular. Desenvolver um simulador completo durante uma fase dissertação de mestrado é uma hipótese totalmente descartada, devido tanto a limitação de tempo hábil como a de recursos humanos. Ainda assim, mesmo na hipótese de um maior número de recursos humanos, este tipo de trabalho muito provavelmente não seria concluído durante este tempo, pois não há uma relação linear entre tempo e recursos humanos, ou seja, uma pessoa desenvolvendo um trabalho durante cinco anos, não equivale necessariamente a cinco pessoas desenvolvendo o mesmo trabalho durante um ano [Fairley 85].

Cientes destes fatores, decidiu-se consagrar esta primeira etapa no desenvolvimento de uma biblioteca básica como base estrutural tanto para um futuro simulador destinando à análise e controle de processos industriais, bem como para qualquer *software* da área de controle, quer seja de uso pessoal e específico, quer seja visando a criação de pacotes ou ambientes computacionais de porte. O desenvolvimento desta biblioteca é fundamentado nas facilidades das estações de trabalho, nas possibilidades oferecidas pelo sistema UNIX® e nos

recursos da linguagem C++. Mais precisamente, esta biblioteca é constituída por tipos abstratos de dados básicos de interesse comum à área de controle de processos.

Outro ponto importante deste trabalho consiste na tentativa de definir uma linha padronizada de desenvolvimento e documentação de rotinas em C++ no LCMI, de modo que a expectativa de vida útil de cada uma destas rotinas seja otimizada. Isto deve-se principalmente ao fato de que grande parte do que é desenvolvido nos ambientes acadêmicos brasileiros acaba perdendo-se, em função de falta de documentação ou de desenvolvimento mal elaborado, por não prever manutenção ou mesmo uma eventual reutilização.

2.3. - Especificações do Projeto

Consoante o propósito de desenvolvimento de uma biblioteca de tipos abstratos de dados básicos para a área de controle de processos, definiu-se um conjunto de especificações, descritas a seguir, para atender este objetivo:

- Criar na forma de módulos de biblioteca, os seguintes tipos abstratos de dados:

  - vetor de números reais,

  - matriz de números reais,

  - vetor de números complexos,

  - matriz de números complexos,

  - polinômios,

  - funções de transferência, e

  - sistemas de equações de estado;

- Realizar a implementação dos tipos de dados previstos em módulos funcionais e bem definidos, principalmente no que concerne a quantidade e qualidade dos métodos (funções) a serem implementados;

- Para cada um destes tipos, propiciar flexibilidade na construção do dado;

- Para facilitar ainda o uso das rotinas, prover meios de entrada e saída de dados, tanto por teclado como por arquivo, principalmente para os tipos mais aprimorados;
- Para cada um dos tipos abstratos, propiciar também uma aritmética mínima;
- Para dar robustez às rotinas e facilitar a depuração de erros, criar um suporte com mensagens de erro, indicando a ocorrência de situação excepcional à rotina;
- Para ainda facilitar a depuração e uso de rotinas, providenciar documentação tanto para utilização como para futuras manutenções destas, numa forma clara, concisa e específica, isto é, cada informação deve estar devidamente localizada, ser necessária e não redundante;
- Também no sentido de tornar mais fácil futuras manutenções, implementações e o uso das rotinas, sugerir então um padrão mínimo de desenvolvimento e documentação de classes em C++ para o LCMI;
- Realizar sempre o uso devido e correto de variáveis, ou seja, trabalhar as variáveis tratando adequadamente o tipo, a precisão e o escopo;
- Para preservar o consumo de memória, usar sempre que possível alocação dinâmica de memória;
- Evitar o uso de desvios incondicionais e, conseqüentemente, a chamada programação "*spaghetti*";
- Dar especial atenção às rotinas que exijam muitas iterações de forma a garantir uma boa eficiência;
- Procurar meios de reutilizar pacotes de rotinas matemáticas consagradas, como por exemplo o LINPACK (Dongarra et alli, 1979) e o EISPACK (Smith et alli, 1976), provendo deste modo economia no esforço de programação; e,
- Usar conceitos e critérios da Engenharia de *Software* que possam aplanar as dificuldades de planejamento e implementação, garantir uma boa qualidade e portanto, melhorar a expectativa de sobrevida das rotinas.

Finalmente, convém salientar que este conjunto de especificações não adota restrições quanto à forma com que os métodos devem ser implementados, justamente para dar uma maior flexibilidade na implementação e também em função da inexperiência inicial sobre a metodologia ideal de como realizar o projeto. Este fato faz com que a fase de implementação ajuste os detalhes das especificações, conforme a aquisição de conhecimento durante o desenrolar do projeto. Outro ponto importante é que especificações flexíveis permitem uma maior segurança no cumprimento do cronograma.

2.4. - Justificativas do Projeto

A seleção de meios para a implementação de um dado projeto, leva em consideração não somente a especificação do produto a obter, mas também como estes meios podem aumentar a probabilidade de se alcançar o objetivo e de se otimizar a qualidade e vida útil do mesmo. No caso de um projeto de mestrado, trabalha-se com limite de tempo fixo (a princípio, de 12 a 18 meses), logo torna-se importante observar que as decisões relativas à seleção dos meios deve considerar aspectos tais como os descritos a seguir:

- Disponibilidade presente e futura de recursos - priorizar recursos já disponíveis em relação aos com apenas uma dada expectativa de chegada;
- Recursos já existentes - priorizar aqueles em maior disponibilidade e, no caso de equipamentos, aqueles com probabilidades maiores de receber manutenção ou pelo menos os que sejam considerados mais confiáveis e robustos; e,
- Atraso e avanço tecnológico - escolher recursos de uso específico que não estejam em fase de saída (extinção) de mercado ou sem uma boa expectativa de entrada (difusão);

No caso deste projeto, os fatores acima descritos têm maior valorização devido ao fato que será elaborado um trabalho de base para outros e, portanto, que deve estar sujeito a uma característica de utilidade de longo prazo.

Para a concretização do projeto então, surgiram basicamente a necessidade de escolha de uma linguagem de programação, um sistema operacional e um tipo de computador para realizar o desenvolvimento. Dentre as possibilidades ao alcance e considerando as

preocupações acima discorridas, estavam linguagens procedurais e orientadas a objetos, os sistemas operacionais MS-DOS®[1] e UNIX®, e computadores do tipo PC, estações de trabalho ou ainda computadores de grande porte do Núcleo de Processamento de Dados da UFSC (NPD - UFSC).

Para a área de desenvolvimento de pacotes computacionais de auxílio ao engenheiro de controle, o LCMI procura dar preferência aos microcomputadores e as estações de trabalho pelo fato destes equipamentos serem largamente disseminados no meio comunitário em geral. O material disponível no início do projeto consistia de microcomputadores do tipo PC, trabalhando sob o sistema operacional MS-DOS® e em sua maioria ligados em rede local e por estações de trabalho do tipo Sun® SPARC, também ligadas em rede local e operando sob o sistema operacional UNIX®.

Quanto aos compiladores restringiu-se a escolha entre o C, o FORTRAN, o PASCAL e o C++, por serem linguagens já consagradas e também bastante difundidas principalmente no meio tecnológico. A figura 2.1., na página seguinte, ajuda a ilustrar esse fato, através de um dos quadros comparativos de uma pesquisa realizada entre 2000 assinantes da revista *IEEE Spectrum* sobre os padrões de utilização de *software* e sobre as necessidades de seus usuários.

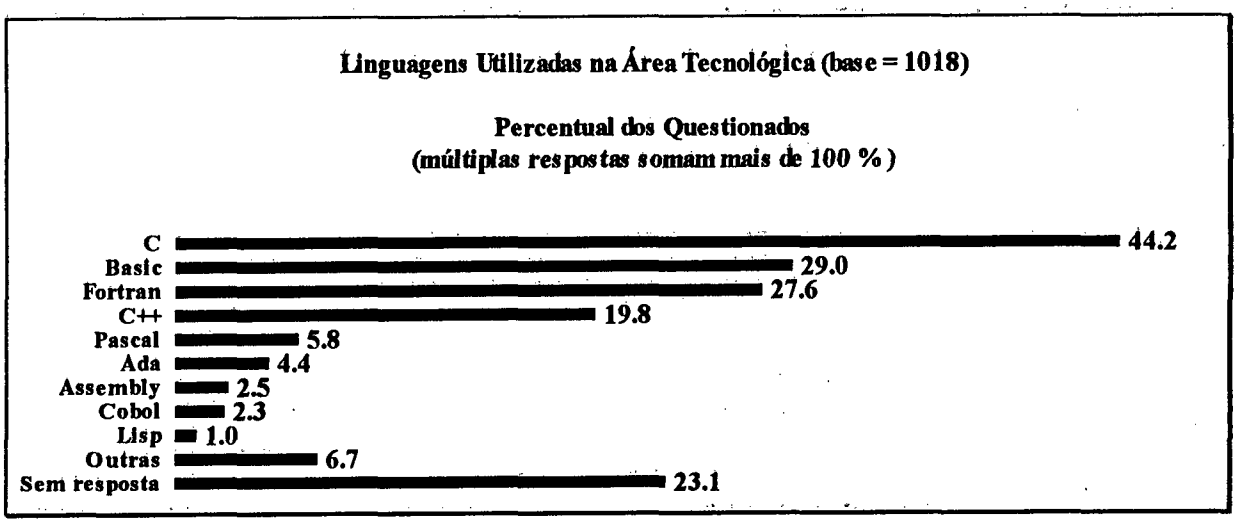


Figura 2.1. - Linguagens Utilizadas na Área Tecnológica[2] *(Fonte: [Kaplan 92]).*

[1] MS-DOS é marca registrada de Microsoft Corporation.
[2] Os dados são baseados em usuários de microcomputadores do tipo PC, de estações de trabalho e de terminais.

2.4.1. - Linguagem C++

Dentre as linguagens citadas - C, FORTRAN, PASCAL e C++ - optou-se pelo paradigma de programação orientado a objetos e, em específico, pela linguagem C++ devido ao conjunto de características apresentadas a seguir:

- difusão a nível mundial - o C++ encontra-se atualmente bastante difundido a nível mundial e, por conseguinte, apresenta uma vasta literatura que contribui para facilitar o aprendizado da mesma;

- eficiência e portabilidade - como é uma linguagem derivada do C, o C++ consegue aproveitar a já conhecida eficiência e portabilidade da linguagem de programação C [Takahashi 88]; e

- semelhança com o C - o conhecimento da linguagem procedural C facilita muito o aprendizado da linguagem C++ e, conseqüentemente, do paradigma de programação orientada a objetos; outrossim, o compilador C++ é plenamente capaz de fazer uso (reutilizar) de rotinas desenvolvidas em C;

- programação orientada a objetos (POO) - a linguagem C++ possibilita a POO e, portanto, permite trabalhar com um paradigma voltado a um conceito bastante difundido pela Engenharia de *Software* como fator preponderante na produtividade e qualidade do desenvolvimento de programas e sistemas - a **reutilização** [Lewis 91], quer seja na forma de herança, quer seja na forma de cliente [Meyer 90];

- abstração de dados e sobrecarga de funções e/ou operadores - existem meios no C++ de prover com simplicidade a abstração de dados necessários ao projeto, assim como de criar ou estender a definição de funções e/ou operadores pré-definidos pelo C/C++, a tal ponto que para o usuário o tratamento dos tipos abstraídos seja muito semelhante aos tipos internos inerentes ao compilador;

- proteção aos membros de uma classe de objetos[1] - é possível definir níveis de acesso - restrições - aos membros (dados e funções) de uma classe,

[1] Basicamente uma classe de objetos consiste em uma unidade que agrupa estrutura de dados e funções associadas.

propiciando desse modo maior robustez na programação [Eckel 91, Sun 89a, Sun 89b e outros]; e

- *interface* e implementação - a existência de arquivos de *interface* e de implementação faz com que um módulo possa ter características mais eficientes em termos de fatores[1] como clareza ou legibilidade, coesão e acoplamento, ocultação de informação (*information hiding*) e inclusive estética, corroborando assim para facilitar a manutenção e a reutilização;

2.4.2. - Sistema Operacional UNIX® e Estações de Trabalho Sun®

Além da linguagem C++, foram selecionados em associação os recursos sistema operacional UNIX® e estações de trabalho Sun® SPARC, apesar da predominância no mercado mundial de microcomputadores do tipo PC em associação ao sistema operacional DOS.

As razões para a escolha deste conjunto fica estabelecida consoante os motivos abaixo descritos:

- Capacidades multiusuário e multitarefa - o conjunto UNIX®/Sun® pode permitir a execução de diversos processos por mais de um usuário ao mesmo tempo [Deitel 84, Montanaro 90] ;
- Desempenho da CPU - as estações apresentam uma arquitetura cujo desempenho de processamento de dados é bastante satisfatório ou melhor em relação a microcomputadores do tipo PC [Montanaro 90];
- Memória disponível - normalmente trabalha-se numa estação com uma disponibilidade de memória bem superior a de microcomputadores PC, de forma que há a possibilidade real do tratamento de problemas de maior grandeza [Montanaro 90];
- Resolução gráfica do monitor - o monitor das estações de trabalho Sun® apresentam excelente resolução gráfica;

[1] Fatores de qualidade de um *software* são amplamente discorridos nas referências [Fairley 85, Pressman 87, Meyer 90].

- Ambiente operacional OpenWindows®[1] - o ambiente OpenWindows® para estações de trabalho facilita consideravelmente a realização de tarefas diversas no computador;

- Disponibilidade de *softwares* de auxílio ao engenheiro - em contraste com a vasta disponibilidade de *softwares* de auxílio ao engenheiro disponíveis para computadores do tipo IBM®-PC[2] , as estações apresentam maior carência destes, seja por motivo de inexistência, seja por dificuldade na aquisição dos mesmos; e,

- Disponibilidade de estações de trabalho - apesar de não ser tão abrangente como o dos microcomputadores PCs, existe um mercado concretizado para estações de trabalho na área tecnológica [Montanaro 90];

2.5. - Etapas do Trabalho

Considerados os objetivos de projeto, algumas linhas de trabalho foram definidas, na forma de etapas a serem cumpridas no decorrer do prazo de, no máximo, 18 meses. O cronograma abaixo, descrevendo as etapas de trabalho, apresenta um prazo de 15 meses.

Etapas de Trabalho	Meses
Revisão bibliográfica	05/92
Estudo dos programas (ambientes e pacotes computacionais) existentes no LCMI	06/92
Estudo básico do sistema operacional UNIX® e do ambiente operacional OpenWindows®	07/92
Estudo aprofundado da linguagem C++	08 a 09/92
Especificação, implementação, verificação, validação e documentação dos módulos	10 a 12/92 e 02 a 03/93

Quadro 2.1. - Cronograma do projeto.

[1] OpenWindows é marca registrada de Sun Microsystems, Inc.
[2] IBM é marca registrada de Industrial Business Machines, Inc.

Etapas de Trabalho	Meses
Sugestão de um padrão de desenvolvimento e documentação de classes C++ para o LCMI	04/93
Testes complementares e documentação definitiva dos módulos	05/93
Redação do relatório final e preparação da apresentação do trabalho	06 a 08/93

Quadro 2.1. - Cronograma do projeto (continuação).

Convém ressaltar que a ordem estabelecida acima não denota uma seqüência rígida de desenrolar do projeto, porquanto retroações a determinadas etapas são sempre necessárias para garantir o aprimoramento e ajuste do projeto.

2.6. - Conclusões

Como pode-se perceber este empreendimento não visa a confecção de um simulador para análise e projeto de processos industriais, mas sim criar uma base estrutural para trabalhos do tipo. A idéia ainda foi estendida no sentido de promover um projeto que não seja específico apenas para um dado *software*, mas que também servisse de utilidade para qualquer outro projeto que necessitasse dos tipos abstratos especificados. Com isso, estabeleceu-se então uma preocupação quanto aos fatores reutilização e manutenção das rotinas e, conseqüentemente, enfoques mais contemporâneos foram estipulados para a realização do trabalho, como por exemplo, maior valorização de etapas tais como a pouco apreciada arte de documentação.

O grau de dificuldade deste projeto foi flexibilizado com a determinação de especificações pouco restritivas, de forma que fosse firmada uma expectativa realística de conclusão do plano de trabalho, nos prazos de um programa de mestrado e consoante os recursos disponíveis.

Convém também salientar que apesar de ser bastante difundida a nível mundial, este tipo de proposta tem um certo pioneirismo no âmbito do LCMI, pois reúne o seguinte conjunto de características: (a) incentivo ao cooperativismo na realização de programas e rotinas (em particular, para a linguagem C++), (b) uso do paradigma orientado a objetos e de conceitos de Engenharia de *Software*, visando facilitar a reutilização e a manutenção, e (c) adequação da

implementação e documentação de classes de objetos em C++ à realidade acadêmica, inclusive com sugestão de um padrão para o laboratório.

2.7. - Referências Específicas

[Deitel 84] DEITEL, Harvey M. An Introduction to Operating Systems. 1. ed. revisada. Massachusetts, USA: Addison-Wesley Publishing Company, 1984.

[Eckel 91] ECKEL, Bruce. C++ Guia do Usuário. Tradução de Luis Antonio Fontes Quintela, revisão técnica de Edison Raymundi Júnior. São Paulo - S.P.: Makron Books do Brasil Editora Ltda. e Editora McGraw-Hill, 1991.

[Fairley 85] FAIRLEY, Richard E. Software Engineering Concepts. Singapore: McGraw-Hill Book Company, 1985.

[Ferreira 87] FERREIRA, P. A. V., FONTANINI, W., GUERRA, A. C., AMARAL, W. C., GOMIDE, F. A. C., Modelagem, Análise e Projeto de Sistemas de Dinâmicos Integrados por Computador. *Revista SBA: Controle & Automação*, v. 1, n. 4, p. 322 - 330, out. 1987.

[Fontanini 90] FONTANINI, W., SILVA Fº., O. S., FERREIRA, P. A. V. Um Ambiente Integrado para Análise e Projeto no Espaço de Estados. *Anais do 8º. CBA - Congresso Brasileiro de Automática*, Belém - P.A., p. 189 - 194, 1990.

[Griss 91] GRISS, Martin L., ADAMS, Sam S., BAETJER Jr., Howard, COX, Brad J., GOLDBERG, Adele. The Economics of Software Reuse. *Proceedings of OOPSLA'91*, Phoenix, Arizona, v. 26, n. 11, p. 264 - 270, maio 1993.

[Kaplan 92] KAPLAN, Gadi. Talking About Tools. *IEEE Spectrum*, v. 29, n. 11, p. 32 - 33, nov. 1992.

[Lewis 91] LEWIS, John A., HENRY, Sallie M., KAFURA, Dennis G., SCHULMAN, Robert S. An Empirical Study of the Object-Oriented Paradigm and Software Reuse. *Proceedings of OOPSLA'91*, Phoenix, Arizona, v. 26, n. 11, p. 184 - 196, nov. 1991.

[Meyer 88] MEYER, Bertrand. Object-oriented Software Construction. Cambridge, Great Britain: Prentice Hall, 1988.

[Montanaro 90] MONTANARO, George D., FREDERICK, Dean K. Workstations as Environments for the Analysis and Design of Controls Systems. *IEEE Control Systems Magazine*, v. 10, n. 3, p. 114 - 121, abr. 1990.

[Pressman 87] PRESSMAN, Roger S. Software Engineering: A Practitioner's Approach. 2. ed. Singapore: McGraw-Hill Book Company, 1987.

[Savi 88] SAVI, Vânio M., CASTELAN N., Eugênio B., BRUCIAPAGLIA, Augusto H., FARINES, Jean-Marie. Uma Visão sobre o Desenvolvimento de Pacotes de Projeto Assistido por Computador para Sistemas de Controle. *Anais do 7º. CBA - Congresso Brasileiro de Automática*, ITA, São José dos Campos - S.P., v. 1, p. 173 - 178, 1988.

[Sun 89a] SUN MICROSYSTEMS. Sun® C++ Programmer's Guide. Mountain View - C.A., USA: Sun Microsystems, Inc., 1989.

[Sun 89b] SUN MICROSYSTEMS, AT&T. AT&T C++ Language System - Reference Manual. USA: AT&T, 1989.

[Takahashi 88] TAKAHASHI, Tadao. Introdução a Programação Orientada a Objetos. Edição EBAI. Curitiba - P.R.: III EBAI, jan. 1988.

CAPÍTULO 3

IMPLEMENTAÇÃO DO PROJETO

3.1. - Introdução

A arte de programação preserva ainda toda sua complexidade de implementação de *software*, cujo objetivo principal seja, em termos gerais, atingir um produto de mercado. Além da suscetibilidade a obstáculos que vão desde a capacitação dos profissionais envolvidos até a disponibilidade de recursos necessários, esta complexidade manifesta-se também em razão da necessidade de se obter uma boa especificação do produto.

Esta problemática de especificação pode ser entendida tanto pela dificuldade de transpor uma idéia para especificações flexíveis, coerentes e claras, como por motivos de desconhecimento do potencial oferecido pelas ferramentas a serem utilizadas, ou ainda pela falsa e freqüente expectativa criada sobre a facilidade de implementação de uma dada rotina.

Para este trabalho em específico, a decisão sobre a metodologia de implementação conta com alguns conceitos fundamentais de projeto. Conceitos como a abstração, a ocultação de informação, a estruturação, a modularidade e, até mesmo, a estética [Fairley 85, Pressman 87]. Juntamente a estes, também destaca-se o uso do paradigma de programação orientada a objetos, assim como aspectos voltados à verificação, à validação e à documentação dos módulos em questão. De fato, todos estes tópicos são favorecidos porquanto podem oferecer facilidades em termos de reutilização e de manutenção.

Outrossim, são acrescentadas especificações, essencialmente voltadas à codificação, criando assim um nível mais detalhado de especificação do projeto. Algumas destas especificações podem, eventualmente, servir como sugestões com o objetivo de padronizar a programação em C++, a nível do Laboratório de Controle e Microinformática da UFSC. Convém todavia ressaltar que tais especificações adicionais não consistem em técnicas de programação, mas simplesmente numa estilística de codificação que, por ventura, ajude a obter maior clareza do código e também facilite a documentação.

A divisão deste capítulo traz, na seção seguinte, o objeto de implementação deste projeto, com referência ao paradigma de programação orientada a objetos e, mais especificamente, em relação à linguagem C++. Na seção 3.3., são discorridos os critérios adotados para a programação propriamente dita; na seção 3.4., aparecem as convenções estipuladas para o projeto e, na seção 3.5., uma sinopse dos padrões de desenvolvimento e documentação elaborados na forma de uma norma interna para o LCMI. A seção 3.6. explica as formas de reutilização de código envolvidas no projeto.

Uma vez compreendido o objeto de trabalho e a estrutura selecionada para tanto, situa-se, na seção 3.7., o escopo do projeto em relação aos sistemas *CACE* (*Computer Aided Control Engineering*). E, por fim, as seções 3.8. e 3.9. apresentam, respectivamente, as conclusões e referências específicas deste capítulo.

3.2. - Objeto de Implementação do Projeto

Trabalhar com o paradigma de programação orientada a objetos, na linguagem C++, significa ter a possibilidade de implementar os tipos abstratos de dados através de classes de objetos. São justamente estas classes de objetos que caracterizam o objeto de implementação deste projeto. Antes porém, é interessante ter uma idéia de como opera o paradigma de programação orientada a objetos.

3.2.1. - Paradigma de Programação Orientada a Objetos

Sucintamente, o paradigma de programação orientada a objetos consiste em uma nova maneira de pensar numa tarefa a ser resolvida com o uso do computador. Ao invés de se tentar adaptar o problema ao computador, o computador é que sofre adaptação ao problema. Na programação orientada a objetos (POO), um problema é examinado para entidades independentes, que se relacionam com outras partes do mesmo. Tais entidades não são escolhidas por sua "computabilidade", mas simplesmente por terem uma fronteira física ou conceitual que as separa do resto do problema. Estas entidades são representadas como objetos no programa de

computador, onde portanto, a meta é ter correspondência, uma a uma, entre entidades no problema físico e objetos no programa [Eckel 91].

Em termos da programação propriamente dita, trabalha-se sobre estruturas de dados encapsuladas em módulos - as chamadas classes de objetos - as quais têm seu acesso a estrutura disciplinado por funções de acessibilidade. A rigor, o paradigma de programação orientado a objetos também prevê facilidades para se trabalhar com herança, polimorfismo, ligação dinâmica e redefinição [Meyer 88].

Do ponto de vista informativo, as linhas gerais destes últimos conceitos podem elucidar a herança como aquela que permite gerar uma classe a partir de outra, simplesmente reaproveitando o esforço de programação, e definindo a classe em questão como uma extensão ou restrição da outra. O polimorfismo, por sua vez, possibilita que uma entidade do programa referencie, em tempo real, objetos de mais de uma classe; e, a ligação dinâmica, por outro lado, permite que diferentes implementações de operações e funções sejam também acessadas em tempo real. Aliás, ambos estes conceitos evidenciam a idéia de uso de *interfaces* idênticas para diferentes implementações. E, por fim, a redefinição consiste justamente nessa capacidade de um método ter mais de uma implementação [Meyer 88, Eckel 91].

3.2.2. - Estrutura Básica das Classes de Objetos em C++

A princípio, poder-se-ia trabalhar o modelo de POO com qualquer linguagem, mas a existência de linguagens específicas para tanto - as denominadas linguagens orientadas a objeto (LOO) - facilitam em muito este tipo de tarefa. Nesse sentido, a opção pela linguagem C++ já contribui para estipular uma forma bastante eficiente e amigável de se implementar os tipos de dados abstratos especificados para este projeto. Esta forma nada mais é do que a já citada classe de objetos. Em C++, cada classe de objetos possui basicamente dados e funções pertinentes, cuja estrutura pode apresentar:

- dados ou atributos - toda classe de objetos pode conter atributos na forma de dados, expressos através de variáveis dos tipos internos ao compilador, estruturas de dados ou mesmo outros tipos abstratos de dados;
- funções membros - toda classe de objetos pode conter funções pertinentes responsáveis pela manipulação dos atributos (ou dados membros) da classe;

- funções amigas - são basicamente funções não membro, para as quais foi concedido acesso especial à classe;
- construtor(es) - uma função construtora consiste numa função especial responsável pela declaração e definição do novo objeto (ou tipo de dado a ser abstraído); e,
- destrutor - uma função destrutora (única), por outro lado, consiste numa função especial usada para automaticamente gerenciar a desalocação de objetos que perderam escopo no programa;

A linguagem C++ permite também que uma classe seja dividida num arquivo de *interface* e noutro de implementação. Isto contribui para:

- facilitar a clareza da classe;
- permitir o teste de rotinas apenas com o uso apenas do arquivo de *interface*;
- dar maior legibilidade aos critérios de acoplamento - relação entre os elementos de diferentes classes - e coesão - relação entre os elementos da mesma classe; e,
- trabalhar sobre uma estrutura eficiente e coerente.

3.2.3. - Níveis de Proteção das Classes de Objetos em C++

Ainda em C++, há um modo de proteger cada dado ou função de uma classe. Os especificadores de proteção ao dado ou à função - *public*, *protected* e *private* - são os responsáveis pelo nível de proteção oferecido. Toda rotina desenvolvida para uma classe, sempre terá acesso aos dados e funções da própria classe. Este nível de proteção, então, serve para controlar a acessibilidade ou de usuário via cliente ou de usuário via herança. O uso de biblioteca tradicionalmente conhecido é o denominado uso via cliente ("comprar recursos") e o uso de uma classe da biblioteca para desenvolver outra com as mesmas características, acrescidas de um detalhamento ou restrição maior, indica o uso via herança e define uma relação classe base/classe herdeira ("herdar recursos") [Meyer 88]. De fato, herdar significa reaproveitar as características de uma classe base e definir outra descendente.

A figura 3.1. ilustra melhor as formas de utilização de uma classe de objetos. Em (a) uma classe **desenho** pode usar as classes **reta, círculo** e **quadrado** (uso via cliente) e em (b) uma classe **polígono** pode originar uma classe **retângulo** que, por sua vez, pode originar uma classe **quadrado**, já que todo retângulo é um polígono e todo quadrado é um retângulo (uso via herança).

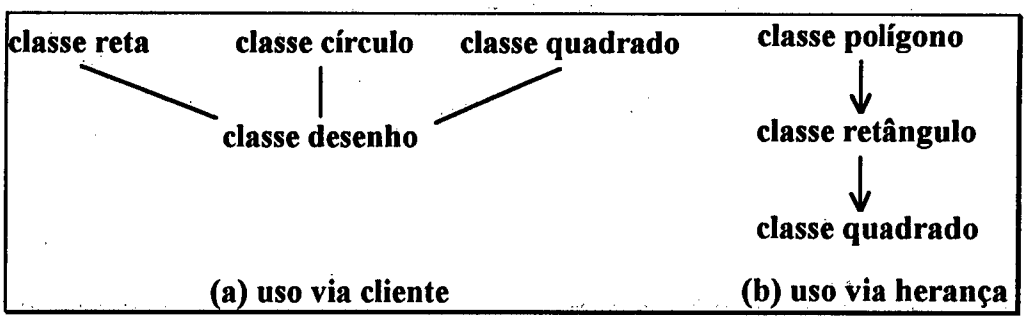


Figura 3.1. - Exemplo de uso de classe de objetos tanto via cliente como via herança.

Uma vez compreendida a forma de uso das classes, então é possível explanar os especificadores de proteção de dados ou funções. O especificador *public* indica que o dado ou função é público e, portanto, pode ser usado, sem nenhuma restrição, tanto via cliente como via herança; o especificador *protected* cria um nível de proteção que não permite o acesso direto pelo usuário cliente, mas garante acesso para outras classes herdeiras da classe em questão; e, o especificador *private* não permite o acesso direto nem por via cliente, nem por via herança, ou seja, somente funções da própria classe podem ter acesso aos dados e/ou funções *private*.

Esses níveis de proteção são justamente destinados a evitar que usuários, inadvertidamente, alterem dados da classe sem o prévio conhecimento dos efeitos colaterais e, por conseguinte, habilitem assim possíveis funcionamentos incorretos da classe de objetos.

3.3. - Critérios Adotados na Programação C++

À medida que este projeto tinha o seu andamento, foram sendo definidos critérios de programação com caracter voltado à programação usando a linguagem C++. Estes critérios de implementação constituem, de fato, especificações adicionais e servem como justaposição ao objetivo de obtenção de um produto de qualidade neste trabalho.

3.3.1. - Especificadores de Proteção

Tendo em vista a funcionalidade dos especificadores de proteção discorridos no final da seção anterior, decidiu-se definir para este projeto, o uso de especificadores de proteção segundo os critérios dados abaixo:

- especificador *public* - usar este especificador para todas as funções que o usuário deva ter acesso e, em contraposição, evitar fortemente qualquer definição de dados públicos;

  Exs.: funções de acesso aos dados e funções de manipulação de dados.

- especificador *protected* - usar este para dados e funções que possam ser herdados por outras classes, porém que não sejam de utilidade para um usuário cliente; e,

  Exs.: atributos da classe de objetos e funções de manipulação interna de dados.

- especificador *private* - usar para dados e funções apenas de uso restrito à própria classe em questão, isto é, sem interesse para clientes ou herdeiros;

  Exs.: funções de auxílio à outras funções principais.

Como pode ser notado, deve-se evitar o uso de dados públicos, pois estes merecem funções de acesso para garantir que um usuário não acesse diretamente os dados e, portanto, não os modifique sem o prévio conhecimento do funcionamento da representação interna da classe (tipo abstrato de dado). Tais funções de acesso devem ter um tratamento de erro quanto à leitura e/ou escrita de um atributo do objeto (instância da classe).

Estes procedimentos em conjunto facilitam o uso das classes, pois os usuários necessitam analisar apenas o arquivo de *interface*, de onde o usuário via cliente só precisa observar as funções do tipo *public* e o usuário via herança somente os dados e funções do tipo *public* e *protected*. Isto garante um encapsulamento de dados ideal para a reutilização de classes e, mesmo para a fase manutenção, o programador deve encontrar uma estrutura bem definida e coerente para realizar a depuração dos métodos da classe.

3.3.2. - Especificador *const*

Como continuidade à idéia de proteção dos dados, surge também o uso do especificador opcional *const*. Além da conhecida possibilidade de definir constantes no programa, há também duas outras formas previstas e recomendadas de proteção a serem utilizadas com este especificador:

- proteção de parâmetro de uma função qualquer - qualquer parâmetro constante de uma função pode levar o especificador *const*, indicando que este não pode ser alterado durante a execução da função; e,

- proteção de função membro propriamente dita - qualquer função membro de uma classe pode receber o especificador *const*, indicando que os dados membros da classe serão considerados constantes durante o escopo da função e, a propósito, este é o único tipo de função membro que pode ser devidamente chamado para um objeto *const*.

Para o último caso - função membro *const* - convém ainda salientar que este tipo de função somente pode chamar outras funções membros *const*. Todavia, os construtores e destrutores não precisam ser declarados funções membros *const*, quando invocados para objetos *const* [Eckel 91].

3.3.3. - Especificador *virtual*

O especificador *virtual* é usado, na linguagem C++, para viabilizar o princípio de polimorfismo. Aliás, a palavra *virtual* indica a existência como faculdade, mas sem exercício ou efeito atual, ou então, suscetibilidade de se realizar; e, polimorfismo significa ter a capacidade de assumir mais de uma forma [Bueno 91]. Nesse sentido e para viabilizar a correta chamada de um destrutor da uma classe, recomenda-se sempre usar o especificador *virtual* para o destrutor de cada classe. Esta medida já antevê a possibilidade do destrutor de uma classe base ser redefinido no destrutor de uma respectiva classe derivada.

Convém todavia ressaltar que nem sempre é necessário redefinir o destrutor de uma classe derivada, pois no caso simples de aprimoramento da representação interna da classe base, deve-se apenas acrescentar o procedimento de desalocação (se houver) da parte aprimorada, pois os compiladores C++ já providenciam uma adequada ordem de chamada de destrutores.

3.3.4. - Especificador *register*

Outra medida opcional, porém importante em termos de eficiência das rotinas é o uso de registradores - especificador de armazenamento *register* - principalmente para variáveis usadas como contador ou similar. Devido ao rápido acesso dos registradores, as rotinas com iterações tendem a ser mais eficientes em termos de velocidade de execução, quando comparadas com e sem o uso dos registradores. Contudo, aconselha-se evitar o abuso destes, pois isto pode acarretar em falta dos mesmos nos momentos realmente necessários.

3.3.5. - Especificador *static*

O uso do especificador de armazenamento *static* pode otimizar o consumo de memória se adequadamente usado. Por exemplo, se um atributo é único para qualquer instância de uma classe, então, este pode ser declarado na *interface* como *static*, reservando assim apenas uma mesma área de memória para todas as instâncias da classe em questão. Entretanto, se usado inadequadamente, pode ter efeito contrário, pois uma variável do tipo *static*, a depender da situação, tem seu escopo desnecessariamente aumentado.

Recomenda-se, outrossim, propiciar a devida atenção às variáveis do tipo *static*, pois estas seguem um princípio de inicialização e armazenamento de dados distinto das variáveis dinâmicas.

3.3.6. - Especificador *friend*

O uso de funções amigas - especificador *friend* - é sempre válido, entretanto, é comum a sugestão no sentido de evitar a sua utilização. Logicamente, nem sempre é possível contornar a necessidade de funções amigas, mas se possível, aconselha-se não usá-las, pois este tipo de função pode tornar a manutenção mais complexa, já que a sua relação (acoplamento) com uma dada classe, às vezes, é pouco clara.

3.3.7. - Estilística de Codificação

Para a programação ainda é prevista uma série de recomendações, algumas já bastante difundidas, porém sempre dignas de serem lembradas, no sentido de tornar mais clara o código-fonte dos programas e, conseqüentemente, facilitar a manutenção. Estas recomendações são:

- utilizar endentação;
- utilizar espaços, linhas em branco e similares, sempre que convenientes para a separação dos blocos de programas;
- utilizar parênteses, principalmente em expressões matemáticas dúbias;
- usar, no máximo, 80 colunas para os programas fontes, porquanto tal medida facilita a impressão e a editoração em terminais;
- criar funções auxiliares para funções consideradas de porte, inclusive evitando assim, endentações profundas e não desejáveis;
- evitar fortemente o uso do comando "*goto*", pois isto caracteriza um desvio incondicional e, portanto, pode acarretar em programação "*spaghetti*";
- evitar funções com um número excessivo de parâmetros;
- evitar sempre efeitos colaterais e, caso não seja possível, providenciar a devida documentação para estes;

- evitar a duplicidade de informações em locais distintos e, conseqüentemente, um possível transtorno de informações conflitantes sobre um mesmo aspecto;
- priorizar o comentário com o uso de duas barras ("// ...") - comentário de linha - em relação ao comentário com o uso de barra e asterisco ("/* ... */") - comentário de bloco - pois este último não pode ser incluído dentro de outro comentário;

3.3.8. - Critérios Extras

E para completar o conjunto de critérios adotados na programação C++, destacam-se também os seguintes tópicos:

- usar apropriadamente a transferência de dados - por cópia, por ponteiro e por referência - já que, normalmente, ganha-se em eficiência se dados simples, como um *int* ou um *double*, forem passados via cópia (trata apenas uma cópia do dado) e dados mais aprimorados, como os *arrays*, forem passados por ponteiro ou por referência (ambos tratam o endereço do dado); a passagem por ponteiro difere da por referência simplesmente porque, na primeira, fica clara a idéia do uso do ponteiro e na segunda, apesar do uso de endereço, há uma similaridade com a transferência por cópia;
- estruturar as classes de objetos com funções, visando um baixo acoplamento e uma alta coesão dos módulos; além disso, minimizar a quantidade de código para as classes, pois classes de tamanhos exagerados podem tornar pouco prático o seu uso para programas de pequeno porte;
- evitar uma possível duplicidade de inclusão de arquivo, causada com o uso de diretivas *#include* no decorrer da compilação e linkagem, através da utilização, no arquivo de *interface* de toda classe, de um identificador de compilação no formato apresentado, conforme ilustra a figura 3.2. da página subseqüente; e,

```
#ifndef <identificador_da_classe>
#define <identificador_da_classe>
classe exemplo {
// ...
};
#endif // <identificador_da_classe>
```

Figura 3.2. - Uso de identificador de classe C++.

- testar toda rotina, sem exceção, realizando a verificação (eliminação dos erros) e validação (satisfação do propósito), contando inclusive e se necessário, com a devida realimentação no desenvolvimento, porquanto tal medida ajuda a evitar uma indesejada propagação de erros.

3.4. - Convenções de Projeto

Algumas convenções de projeto também foram definidas, no sentido de promover um tratamento mais aprimorado para as classes, principalmente no que concerne à documentação das mesmas.

- para o compilador Sun® C++ 2.1, recomenda-se que os arquivos de implementação usem a extensão ".cc", pois isto contribui para a diferenciação de arquivos C dos C++;

- uso da língua inglesa para a programação (nomes de variáveis, funções, mensagens de erro e similares) visando uma padronização com a língua normalmente versada pelos compiladores C++;

- uso da língua portuguesa para a documentação, quer seja na forma de comentários no programa (sem caracteres acentuados ou cedilha), quer seja na forma de manuais de utilização; essa documentação pode, se necessário, ser mais facilmente traduzida para a língua inglesa (língua considerada como um padrão mundial) do que partes do código do programa como, por exemplo, os nomes das funções;

- para a manutenção, prevê-se que a documentação esteja inserida como comentários no próprio corpo do programa, onde estes devem ser sucintos, claros e fundamentalmente necessários, já que comentários redundantes apenas minoram a clareza do programa;

- para a utilização, prevê-se manuais impressos e também de consulta rápida (*"man pages"*), sendo este último específico para as estações de trabalho.

3.5. - Padrões de Desenvolvimento e Documentação

Para a difusão, a nível de LCMI, dos padrões de desenvolvimento e de documentação de classes C++, decidiu-se elaborar, na forma de uma norma interna, um conjunto de recomendações para tanto [Suguieda 93a]. No anexo I encontra-se uma cópia da mesma.

Esta norma foi feita com base nos formatos de manuais da linguagem C++, a disposição no mercado, assim como embasada igualmente na experiência adquirida e na exposta por engenheiros do laboratório. E uma vez que seu uso é destinado ao ambiente acadêmico, esta norma preza pela simplicidade e facilidade de realização. Dessa forma, gera-se para o laboratório uma contribuição à padronização de desenvolvimento sustentável para a linguagem C++ e, também, procura-se contornar uma grande resistência à elaboração de documentação por parte dos programadores em geral.

Sucintamente, esta norma interna apresenta as seguintes características:

- definição de um formato base de arquivo *header* (arquivo de *interface* - *.h);

- definição de um formato base de arquivo de corpo (arquivo de implementação - *.cc)

- definição de um formato base de arquivo "*makefile*" (arquivo de gerenciamento de compilação e linkagem);

- recomendações gerais para facilitar o tratamento das classes C++;

- elaboração de uma máscara para um manual de utilização impresso ("Manual de Utilização de Classe C++");

- definição de um formato para manuais de consulta rápida *on-line* para as estações de trabalho ("*man pages*");

- estipulação de um critério de organização das classes, quer seja para a localização e uso, quer seja para o desenvolvimento e catalogação;

Convém ressaltar que para todos os formatos de arquivo citados acima, foram colocados a disposição arquivos exemplos ou mesmo arquivos do tipo máscara (arquivo do tipo "preencha as lacunas"), no intuito de facilitar ainda mais o trabalho de desenvolvimento e documentação. Estes formatos podem, inclusive, ser observados na norma contida no anexo I. No anexo II, pode ser encontrado um exemplo de um "Manual de Utilização de Classe C++" e, no anexo III, um exemplo de como aparecem as "*man pages*" destinadas às estações de trabalho.

3.6. - Reutilização de Código

Uma das abordagens também trabalhadas foi a de estudar meios de reutilizar rotinas fossem elas desenvolvidas no LCMI ou de domínio público, tais como os pacotes LINPACK (Dongarra et alli, 1979) e o EISPACK (Smith et alli, 1976). Estes pacotes foram implementados em FORTRAN e, a princípio, não se adequam com a linguagem C++. Quanto às rotinas do LCMI que ofereciam algum interesse ao projeto, estas apareciam basicamente implementadas nas linguagens FORTRAN, PASCAL e C++.

Primeiramente, para as rotinas em FORTRAN, existe uma compatibilidade entre os compiladores Sun® FORTRAN e Sun® C [Sun 90b]; este último, por sua vez, podendo ser compatibilizado com o compilador Sun® C++ [Sun 89a]. A idéia inicial consistia em tentar uma conexão entre o FORTRAN e o C++, porém devido à forma com que os arquivos são trabalhados nestas compatibilizações, não foi possível concretizar esta intenção, desprovidos do auxílio ou de um pré-compilador ou de um tradutor automático. De fato, o estágio FORTRAN-C não se adequa ao C-C++, porque o primeiro exige a linkagem com o compilador FORTRAN e o segundo, simplesmente, não o aceita.

A solução apareceu com a obtenção, via protocolo *FTP*[1], de um tradutor automático de FORTRAN para C/C++. Este tradutor, denominado f2c[2], gera automaticamente um novo código-fonte em C/C++, a partir do código-fonte em FORTRAN. A propósito, a forma estruturada com que as rotinas dos pacotes LINPACK e EISPACK foram implementados viabilizou o uso deste.

Já para as rotinas do LCMI, com exceção daquelas implementadas em C++, constatou-se que não havia a possibilidade de reutilização direta das mesmas, quer seja por problemas de documentação, quer seja por problemas de estruturação. Isto é, algumas rotinas simplesmente deixavam seu objetivo e/ou funcionamento obscuros, assim como outras apresentavam um forte acoplamento entre rotinas de cálculo e rotinas de entrada e saída de dados, impossibilitando deste modo a completa utilização delas.

Assim sendo, a solução foi realizar a tradução e adaptação manual ou, em muitos casos, a implementação pura e simples das rotinas consideradas válidas de reaproveitar. Com este tipo de atitude, surgiu o problema de delimitar quando a tradução é mais simples que a criação de novas rotinas, principalmente porque este tipo de tarefa, freqüentemente, envolve tanto a compreensão do algoritmo quanto da lógica de outrem. Coube portanto ao bom senso e a experiência, a prerrogativa de julgar tais situações.

3.7. - Escopo do Projeto

O propósito de desenvolvimento de uma biblioteca de tipos básicos, destinados à área de controle de processos, juntamente com a forma decidida de implementação, permitem situar o trabalho na linha de desenvolvimento de sistemas *CACE* (*Computer Aided Control Engineering*). Conceitualmente, um sistema *CACE* pode ser entendido como um ambiente de programação, onde parte significativa da teoria de controle encontra-se descrita através de algoritmos computacionais que podem ser facilmente utilizados e combinados de modo a simplificar tarefas mecânicas e repetitivas de um usuário qualquer do sistema [Ferreira 87].

[1] *FTP* é a abreviação de *File Transfer Protocol*.

[2] f2c é, conforme a sua documentação, um conversor de FORTRAN para C sob o desenvolvimento de David Gay (AT&T Bell Labs), Stu Feldman (Bellcore), Mark Maimone (Carnegie-Mellon University) e Norm Schryer (AT&T Bell Labs).

Os sistemas *CACE* modernos normalmente são constituídos por uma estrutura funcional onde destacam-se:

i) **Base de Utilidades** - um conjunto de rotinas matemáticas básicas (operações algébricas com números complexos, polinômios, matrizes, etc.) as quais suportam o desenvolvimento da **Base de Métodos**;

ii) **Base de Métodos** - um conjunto de procedimentos de modelagem, análise e/ou projeto descritos em termos de algoritmos computacionais que definem, juntamente com a **Base de Dados**, o escopo dentro do qual o sistema trabalha;

iii) **Base de Dados** - uma estrutura de dados flexível que garante uma representação interna eficiente do segmento da teoria considerado, facilmente manipulada pelo usuário;

iv) ***Interface* Homem-Máquina** - meio de comunicação do usuário com o sistema responsável pela entrada e saída de informações, inclusive em forma gráfica, e geralmente associada a uma linguagem orientada para controle, através do qual o usuário pode facilmente expressar as ações que devam ser desenvolvidas e informações que devam ser passadas pelo sistema; e,

v) **Supervisor/Interpretador** - responsável pela coordenação de utilização dos recursos do sistema, particularmente no que se refere ao uso e interpretação dos comandos da linguagem, alocação de memória, manipulação de arquivos, etc.

A figura 3.3., na página subseqüente, apresenta uma visão simplificada de um sistema *CACE* estruturado como descrito em (i) - (v).

Portanto, consoante a estrutura funcional descrita e de acordo com as classes selecionadas para este projeto, entende-se que o trabalho envolve três bases de um sistema *CACE* - a base de dados, a de métodos e a de utilidades. A rigor, o projeto envolve a criação de uma base de utilidades e a estruturação de facilidades para a definição das bases de métodos e de dados, pois os tipos definidos apresentam todo um conjunto de rotinas matemáticas e de engenharia de controle, formando uma base de utilidades, assim como já apresentam métodos de análise para a geração da base de métodos e, também, uma representação de tipos de dados eficiente e flexível para a composição de uma base de dados.

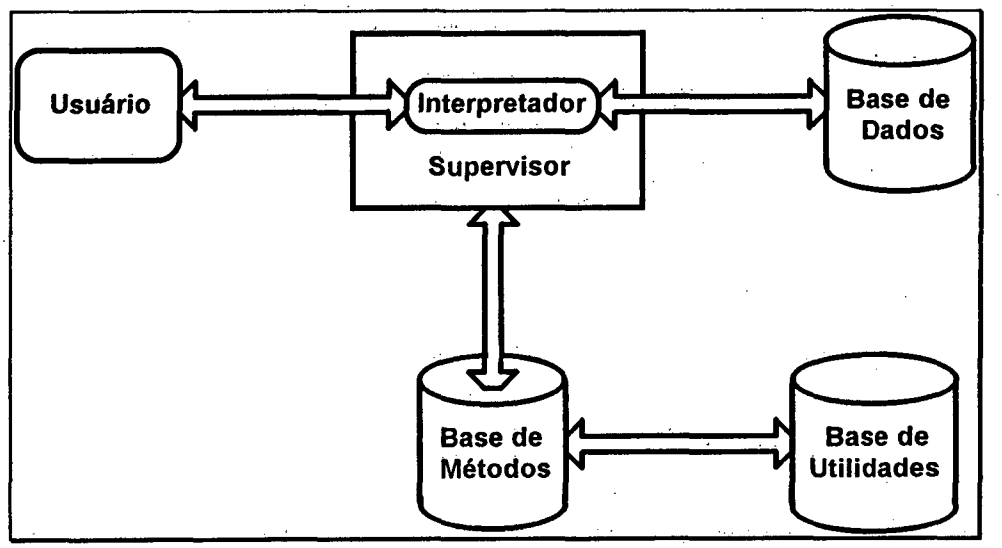


Figura 3.3. - Estrutura simplificada de um sistema *CACE*.

3.8. - Conclusões

Este capítulo abrange, juntamente com o anterior, uma visão de toda a estrutura sob a qual foi desenvolvido o projeto. Inicialmente, situou-se o objeto de implementação do trabalho - as classes de objetos - onde foi apresentado um compêndio de sua estrutura e funcionalidade. Aliás, estas classes compõem e, ao mesmo tempo, são sinônimos para os tipos abstratos de dados. Pôde-se ainda ser observado que tais estruturas são perfeitamente aceitáveis para implementar os tipos abstratos de dados especificados para este projeto, já que a linguagem C++ e o paradigma de programação orientado a objetos possibilitam facilidades para tanto.

Contudo, devido à liberdade de execução e à gama de recursos que a linguagem e o paradigma oferecem, foram explanados uma série de detalhes de implementação, que vão desde recomendações sobre o uso de especificadores da linguagem C++, passando por uma estilística de codificação, chegando até um padrão convencionado das línguas a serem utilizadas para a codificação e documentação dos módulos. Estes detalhes servem como especificações adicionais e, portanto, destinam-se a contentar os objetivos do projeto.

Foram mostradas ainda as soluções encontradas para aplicar a reutilização no projeto, pois o desenvolvimento dos tipos de dados prescritos envolve a necessidade de uma série de algoritmos, muitos dos quais implementados anteriormente. Confirmou-se, além disso, que a

reutilização é bastante difícil de ser aplicada quando se trabalha com módulos que não prevêem este tipo de situação.

Finalmente, estabeleceu-se a ligação da biblioteca de tipos abstratos de dados com os sistemas *CACE*, de modo que fosse possível compreender a sua utilidade para tais sistemas ou para sistemas similares. Na verdade, pretendeu-se com isso demonstrar a consistência do material de base implementado e sua aplicabilidade para futuros *softwares* destinados a área de controle de processos e áreas afins.

3.9. - Referências Específicas

[Bueno 91] BUENO, Francisco da Silveira. Minidicionário da Língua Portuguesa. 5. ed. atualizada. São Paulo - S.P.: Editora Lisa Ltda, 1991.

[Eckel 91] ECKEL, Bruce. C++ Guia do Usuário. Tradução de Luis Antonio Fontes Quintela, revisão técnica de Edison Raymundi Júnior. São Paulo - S.P.: Makron Books do Brasil Editora Ltda. e Editora McGraw-Hill, 1991.

[Fairley 85] FAIRLEY, Richard E. Software Engineering Concepts. Singapore: McGraw-Hill Book Company, 1985.

[Ferreira 87] FERREIRA, P. A. V., FONTANINI, W., GUERRA, A. C., AMARAL, W. C., GOMIDE, F. A. C., Modelagem, Análise e Projeto de Sistemas de Dinâmicos Integrados por Computador. *Revista SBA: Controle & Automação*, v. 1, n. 4, p. 322 - 330, out. 1987.

[Meyer 88] MEYER, Bertrand. Object-oriented Software Construction. Cambridge, Great Britain: Prentice Hall, 1988.

[Pressman 87] PRESSMAN, Roger S. Software Engineering: A Practitioner's Approach. 2. ed. Singapore: McGraw-Hill Book Company, 1987.

[Suguieda 93a] SUGUIEDA, Márcio Heidi, KAMMER, Leonardo César. Norma para Documentação de Classes C++. *Publicação Interna LCMI PI 93-1*, LCMI - UFSC, Florianópolis - S.C., jun. 1993.

CAPÍTULO 4

RESULTADOS DO PROJETO

4.1. - Introdução

Consoante o objetivo maior de planejamento e desenvolvimento de tipos abstratos de dados básicos, destinados à área de controle de processos, foram implementadas algumas classes de objetos definindo os tipos especificados - vetores de números reais e de números complexos, matrizes de números reais e de números complexos, polinômios, funções de transferência e sistemas de equações de estado. De fato, tais tipos estão descritos em nove classes cujos especificadores receberam os seguintes nomes:

- *dvector* - vetores de números reais de dupla precisão (tipo *double*);
- *dmatrix* - matrizes de números reais de dupla precisão (tipo *double*);
- *extdmat (extended dmatrix)* - matrizes estendidas (ou aprimoradas) de números reais de dupla precisão (tipo *double*);
- *cvector* - vetores de números complexos (tipo *complex*);
- *cmatrix* - matrizes de números complexos (tipo *complex*);
- *extcmat (extended cmatrix)* - matrizes estendidas (ou aprimoradas) de números complexos (tipo *complex*);
- *polynom* - polinômios com coeficientes reais e com potências dos termos positivas inteiras;
- *trfunct* - funções de transferência; e,
- *statvar* - sistemas de equações de estado lineares e invariantes no tempo.

Todas as classes implementadas para este projeto obedeceram um conjunto de especificações descritas nos capítulos anteriores. Logo, considera-se não ser mais necessário justificar o porquê do formato das classes adotado. Apenas serão descritos no decorrer deste capítulo sinopses das classes implementadas, as quais denotam a realização do propósito pretendido. Além disso, serão apresentados alguns exemplos ilustrativos.

Assim sendo, este capítulo mostrará nas seção de 4.2. a 4.11., as nove classes implementadas para este projeto. Na seção 4.12. aparece o conjunto de exemplos ilustrativos e, por fim, nas seções 4.13. e 4.14. encontram-se, respectivamente, as conclusões e referências específicas ao capítulo.

4.2. - Aspecto Geral das Classes Implementadas

Todas as classes implementadas possuem um conjunto comum de características, abaixo resumidas, as quais obedeceram as exigências das especificações estabelecidas:

- vários construtores de tipo para cada classe de objetos, visando flexibilidade na criação das instâncias (ou objetos);
- destrutor virtual;
- representação interna operando com alocação dinâmica de memória;
- funções de entrada e saída de dados tanto via *console* (teclado + monitor) como via arquivo *ASCII*; inclusive, com o uso de sobrecarga[1] de operadores >> (entrar em) e << (sair de), os quais permitiram uma flexibilidade na leitura e/ou escrita de dados, pois existe a facilidade sem, no entanto, uma restrição a um padrão específico de entrada e saída de dados;
- funções e operadores pertinentes para manipulação e/ou análise de dados;

[1] Sobrecarga é o termo usado na linguagem C++ para indicar a redefinição de uma função (ou operador), isto é, há a possibilidade de trabalhar um mesmo nome de função ou símbolo de operador para mais de uma implementação.

- sobrecarga de operadores aritméticos básicos (=, +, -, *, /, ==, !=, etc.), de acordo com a necessidade de cada classe; e,
- tratamento de erro, quer seja por simples aviso de problemas, quer seja por interrupção do programa, indicando a razão de uso incorreto da classe.

No decorrer da seção seguinte, aparecem breves sinopses das classes, extraídas com base nos respectivos manuais de utilização elaborados para estas [Suguieda 93b até Suguieda 93j]. Cada um destes manuais, denominados "Manuais de Utilização de Classe C++", estão catalogados no LCMI e apresentam uma documentação completa para a utilização das mesmas.

4.3. - Classe *dvector*

4.3.1. - Sinopse da Classe

A classe *dvector* é a responsável pela abstração de dados do tipo vetor de números reais de dupla precisão (números do tipo *double*). Esta classe foi criada para atender a necessidade de vetores reais alocados dinamicamente na linguagem C++, que exijam recursos de entrada e saída de dados, assim como de uma aritmética básica. Além disso, este tipo serve como base do tipo matriz de números reais de dupla precisão.

O usuário da classe *dvector* praticamente não sentirá diferença ao trabalhar com este tipo de vetor, pois o manuseio deste é similar ao manuseio dos tipos internos ao compilador C++. Sucintamente, o acesso aos elementos dos vetores continua sendo feito pelo operador [] (colchetes); o uso dos operadores =, +, -, *, /, == e != também é válido; a entrada e saída de dados pode ser feita com os operadores >> (entrar em) e << (sair de), tanto por meio do *console* (teclado + monitor) como por meio de arquivos, ou através de funções de acesso aos dados internos (atributos) da classe; e ainda, para dar uma melhor flexibilidade de declaração e inicialização de variáveis, foram concebidos quatro construtores de tipo.

4.3.2. - Funções da Classe

4.3.2.1. - Construtores e Destrutor

Construtores/Destrutor	Propósito
dvector();	declarar uma variável como do tipo *dvector*
dvector(int size, double initvalue = 0.);	declarar e inicializar uma variável *dvector* com um dado tamanho e com todos elementos iguais (opcionalmente zeros)
dvector(int size, double initvalues);*	declarar e inicializar uma variável *dvector* a partir de um *array* (estático ou dinâmico)
dvector(const dvector& argm);	declarar e inicializar um *dvector* a partir de outro e garantir, quando necessária, a conversão adequada de tipos
virtual ~dvector();	(destrutor da classe)

Quadro 4.1. - Construtores e destrutor da classe *dvector*.

4.3.2.2. - Funções e Operadores de Entrada e Saída de Dados

Funções/Operadores	Propósito
friend istream& operator>> (istream& in, dvector& argm);	propiciar a entrada de dados via *console*
friend ifstream& operator>> (ifstream& fin, dvector& argm);	propiciar a entrada de dados via arquivo ASCII
friend ostream& operator<< (ostream& out, const dvector& argm);	propiciar a saída de dados via *console*
friend ofstream& operator<< (ifstream& fout, const dvector& argm);	propiciar a saída de dados via arquivo ASCII
int size() const;	fornecer o tamanho do *dvector*
double& operator[] (int index) const;	permitir a leitura e escrita de elemento no vetor
void setval(int index, double value);	permitir a escrita de elemento no vetor (em posições existentes ou não)

Quadro 4.2. - Funções e operadores de entrada e saída de dados da classe *dvector*.

4.3.2.3. - Demais Funções e Operadores

Funções/Operadores	Propósito
dvector operator= (const dvector& rval);	realizar a atribuição de vetores
dvector operator+ (const dvector& rval) const; *friend dvector operator+ (const dvector& lval, const double rval);* *friend dvector operator+ (const double lval, const dvector& rval);*	realizar a soma de vetores ou a soma de vetor com escalar (apesar desta última não ser matematicamente definida)
dvector operator- (const dvector& rval) const; *dvector operator- (const double rval) const;* *dvector operator- () const;*	realizar ou a subtração de vetores, ou a subtração de vetor por escalar (apesar de não ser matematicamente definida) ou o unário menos
friend dvector operator (const dvector& lval, const double rval);* *friend dvector operator* (const double lval, const dvector& rval);*	realizar a multiplicação de vetor por escalar
dvector operator/ (const double rval) const;	realizar a divisão de vetor por um escalar
int operator== (const dvector& rval) const;	propiciar a comparação de igualdade de vetores
int operator!= (const dvector& rval) const;	propiciar a comparação de desigualdade de vetores

Quadro 4.3. - Demais funções e operadores da classe *dvector*.

4.4. - Classe *dmatrix*

4.4.1. - Sinopse da Classe

Seguindo a mesma linha da classe *dvector*, a classe *dmatrix* é responsabilizada pela abstração de dados do tipo matriz de números reais de dupla precisão (números do tipo *double*). Esta classe foi criada para atender a necessidade de matrizes reais alocadas dinamicamente na linguagem C++, que exijam recursos de entrada e saída de dados, assim como de uma aritmética básica, contando inclusive com as funções de cálculo de transposta, determinante e inversa.

Outrossim, o usuário da classe *dmatrix* praticamente não sentirá diferença ao trabalhar com este tipo de matriz, pois a manipulação desta também é similar a manipulação dos tipos internos ao compilador C++. Basicamente, o acesso aos elementos dos vetores continua sendo feito pelos operadores [][] (colchetes); o uso dos operadores =, +, -, *, /, == e != também é permitido; a entrada e saída de dados pode ser feita com os operadores >> (entrar em) e << (sair de), tanto por meio do *console* (teclado + monitor) como por meio de arquivos, ou através de funções de acesso aos dados internos (atributos) da classe; e ainda, para dar uma melhor flexibilidade de declaração e inicialização de variáveis, foram concebidos seis construtores de tipo.

4.4.2. - Funções da Classe

4.4.2.1. - Construtores e Destrutor

Construtores/Destrutor	Propósito
dmatrix();	declarar uma variável como do tipo *dmatrix*
dmatrix(int mrows, int mcols = 1, double initvalue = 0.);	declarar e inicializar uma variável *dmatrix* com um dado número de linhas e de colunas (opcionalmente uma coluna) e com todos elementos iguais (opcionalmente zeros)
dmatrix(int mrows, int mcols, double initvalues);*	declarar e inicializar uma variável *dmatrix* a partir de um *array* (estático ou dinâmico), o qual deve ser arranjado por linhas
dmatrix(int mrows, dvector initvectors);*	declarar e inicializar uma variável *dmatrix* a partir de um *array* (estático ou dinâmico) de *dvectors* e supondo-os como vetores-linha
dmatrix(char flag, int dimension);*	declarar e inicializar uma variável *dmatrix* como uma matriz identidade
dmatrix(const dmatrix& argm);	declarar e inicializar uma *dmatrix* a partir de outra e garantir, quando necessária, a conversão adequada de tipos
virtual ~dmatrix();	(destrutor da classe)

Quadro 4.4. - Construtores e destrutor da classe *dmatrix*.

4.4.2.2. - Funções e Operadores de Entrada e Saída de Dados

Funções/Operadores	Propósito
friend istream& operator>> (istream& in, dmatrix& argm);	propiciar a entrada de dados via *console*
friend ifstream& operator>> (ifstream& fin, dmatrix& argm);	propiciar a entrada de dados via arquivo ASCII
friend ostream& operator<< (ostream& out, const dmatrix& argm);	propiciar a saída de dados via *console*
friend ofstream& operator<< (ifstream& fout, const dmatrix& argm);	propiciar a saída de dados via arquivo ASCII
int rows() const;	fornecer o número de linhas da *dmatrix*
int cols() const;	fornecer o número de colunas da *dmatrix*
dvector& operator[] (int index) const;	permitir o acesso a vetores da matriz, e em conjunto com o operador [] da classe *dvector*, permite leitura ou escrita de elemento na matriz
void setval(int rowindex, int colindex, double value);	permitir a escrita de elemento na matriz (em posições existentes ou não)
void setrow(int rowindex, dvector vvalue);	permitir a escrita, em uma linha existente ou não, de um *dvector* (assumindo-o como vetor-linha) na matriz
void setcol(int colindex, dvector vvalue);	permitir a escrita, em uma coluna existente ou não, de um *dvector* (assumindo-o como vetor-coluna) na matriz
void copyrow(const dmatrix& from, int from_row, int this_row);	copiar uma linha de uma matriz para outra
void copycol(const dmatrix& from, int from_col, int this_col);	copiar uma coluna de uma matriz para outra

Quadro 4.5. - Funções e operadores de entrada e saída de dados da classe *dmatrix*.

4.4.2.3. - Demais Funções e Operadores

Funções/Operadores	Propósito
void switrow(int row1, int row2);	realizar a simples permuta de linhas de uma matriz
void switcol(int col1, int col2);	realizar a simples permuta de colunas de uma matriz
dmatrix operator= (const dmatrix& rval);	realizar a atribuição de matrizes
dmatrix operator+ (const dmatrix& rval) const; *friend dmatrix operator+ (const dmatrix& lval, const double rval);* *friend dmatrix operator+ (const double lval, const dmatrix& rval);*	realizar a soma de matrizes ou a soma de matriz com escalar (apesar desta última não ser matematicamente definida)
dmatrix operator- (const dmatrix& rval) const; *dmatrix operator- (const double rval) const;* *dmatrix operator- () const;*	realizar ou a subtração de matrizes, ou a subtração de matriz por escalar (apesar de não ser matematicamente definida) ou o unário menos
dmatrix operator (const dmatrix& rval) const;* *friend dmatrix operator* (const dmatrix& lval, const double rval);* *friend dmatrix operator* (const double lval, const dmatrix& rval);*	realizar a multiplicação de matrizes ou de matriz por escalar
dmatrix operator/ (const double rval) const;	realizar a divisão de matriz por um escalar
int operator== (const dmatrix& rval) const;	propiciar a comparação de igualdade de matrizes
int operator!= (const dmatrix& rval) const;	propiciar a comparação de desigualdade de matrizes
double mmin() const;	obter o elemento mínimo (menor valor) da matriz
double mmax() const;	obter o elemento máximo (maior valor) da matriz
double mean() const;	obter a média dos elementos da matriz
double variance() const;	obter a variância populacional dos elementos da matriz
double determinant() const;	obter o determinante da matriz
dmatrix transpose() const;	obter a matriz transposta
dmatrix inverse() const;	obter a matriz inversa

Quadro 4.6. - Demais funções e operadores da classe *dmatrix*.

4.5. - Classe *extdmat*

4.5.1. - Sinopse da Classe

A classe *extdmat* - *extended dmatrix* - foi desenvolvida como um aprimoramento da classe *dmatrix*. Este aprimoramento foi realizado utilizando-se os recursos de herança da linguagem C++, que por conseguinte, permitiram preservar para esta classe todos os recursos da classe base *dmatrix*.

Dentre os recursos mantidos, estão os seis construtores bases, os operadores de entrada (>>) e saída (<<) de dados , as funções (e operadores) de acesso ou de simples manipulação algébrica dos elementos da matriz (*rows, cols, operator [], setval, setrow, setcol, copyrow, copycol, switrow e switcol*), os operadores matemáticos (=, +, -, *, /, == e !=) e as funções matemáticas básicas (*mmin, mmax, mean, variance, determinant, transpose e inverse*).

Para esta classe, foram acrescentados alguns recursos mais específicos à área de sistemas de controle e áreas afins. Ou seja, estão disponíveis também para esta classe as funções *rank* - cálculo do posto, *eigenvalues* - cálculo de autovalores tanto do problema $Ax = \lambda x$ como do problema generalizado $Ax = \lambda Bx$, e *eigenvalvec* - cálculo de autovalores e autovetores associados tanto do problema $Ax = \lambda x$ como do problema generalizado $Ax = \lambda Bx$. Além destes novos recursos, também aparece um novo construtor responsável pela conversão (silenciosa) do tipo *dmatrix* para o tipo *extdmat*, de modo que o uso do tipo *dmatrix*, quando se trabalha com o tipo *extdmat*, é plenamente possível.

4.5.2. - Funções da Classe

4.5.2.1. - Construtores e Destrutor

Construtores/Destrutor	Propósito
extdmat();	declarar uma variável como do tipo *extdmat*, utilizando um construtor da classe base *dmatrix*
extdmat(int mrows, int mcols = 1, double initvalue = 0.);	declarar e inicializar uma variável *extdmat* com um dado número de linhas e de colunas (opcionalmente uma coluna) e com todos elementos iguais (opcionalmente zeros), utilizando um construtor da classe base *dmatrix*
extdmat(int mrows, int mcols, double initvalues);*	declarar e inicializar uma variável *extdmat*, utilizando um construtor da classe base *dmatrix*, a partir de um *array* (estático ou dinâmico), o qual deve ser arranjado por linhas
extdmat(int mrows, dvector initvectors);*	declarar e inicializar uma variável *extdmat*, utilizando um construtor da classe base *dmatrix*, a partir de um *array* (estático ou dinâmico) de *dvectors* e supondo-os como vetores-linha
extdmat(char flag, int dimension);*	declarar e inicializar uma variável *extdmat* como uma matriz identidade, utilizando um construtor da classe base *dmatrix*
extdmat(const extdmat& argm);	declarar e inicializar uma *extdmat*, utilizando um construtor da classe base *dmatrix*, a partir de outra e garantir, quando necessária, a conversão adequada de tipos
extdmat(const dmatrix& argm);	declarar e inicializar uma *extdmat* a partir de uma *dmatrix* ou garantir, quando necessária, a conversão silenciosa do tipo *dmatrix* no tipo *extdmat*
virtual ~extdmat();	(destrutor da classe)

Quadro 4.7. - Construtores e destrutor da classe *extdmat*.

4.5.2.2. - Funções e Operadores de Entrada e Saída de Dados

Esta classe simplesmente herda todas as funções e operadores de entrada e saída de dados já disponíveis na classe base *dmatrix*. Para conhecer estas, basta consultar o quadro 4.5. deste capítulo.

4.5.2.3. - Demais Funções e Operadores

Antes de situar as outras funções e operadores específicas para esta classe, convém salientar que as descritas no quadro 4.6. são nesta também válidas. Segue então, o conjunto restante dos métodos desta classe:

Funções/Operadores	Propósito
int rank(double tol = 0.) const;	calcular o posto da matriz
int eigenvalues(extdmat& eigval) const;	calcular os autovalores da matriz no problema $Ax = \lambda x$
int eigenvalvec(extdmat& eigval, extdmat& eigvec) const;	calcular os autovalores e autovetores associados da matriz no problema $Ax = \lambda x$
int eigenvalues(extdmat& B, extdmat& eigval) const;	calcular os autovalores da matriz no problema generalizado $Ax = \lambda Bx$
int eigenvalvec(extdmat& B, extdmat& eigval, extdmat& eigvec) const;	calcular os autovalores e autovetores associados da matriz no problema generalizado $Ax = \lambda Bx$

Quadro 4.8. - Demais funções e operadores da classe *extdmat*.

4.6. - Classe *cvector*

4.6.1. - Sinopse da Classe

A classe *cvector*, por sua vez, é a responsável pela abstração de dados do tipo vetor de números complexos (tipo *complex*). Esta classe foi criada para atender a necessidade de exijam recursos de entrada e saída de dados, assim como de uma aritmética básica. Além disso, este tipo serve como base do tipo matriz de números complexos.

Mais uma vez, o usuário da classe *cvector* praticamente não sentirá diferença ao trabalhar com este tipo de vetor, pois o manuseio deste é similar ao manuseio dos tipos internos ao compilador C++. O acesso aos elementos dos vetores continua sendo feito pelo operador [] (colchetes); o uso dos operadores =, +, -, *, /, == e != é também válido; a entrada e saída de dados pode ser feita com os operadores >> (entrar em) e << (sair de), tanto por meio do *console* (teclado + monitor) como por meio de arquivos, ou através de funções de acesso aos dados

internos (atributos) da classe; e ainda, para dar uma melhor flexibilidade de declaração e inicialização de variáveis, foram concebidos seis construtores de tipo.

É salientado, outrossim, que os números complexos - tipo *complex* - destinados ao compilador Sun® C++ 2.1 são igualmente tipos abstratos de dados providos, na forma de uma biblioteca, pela AT&T. Logo, recomenda-se aos usuários da classe *cvector*, uma leitura prévia dos manuais [Sun 89a, Sun 89d] para, então, permitir uma melhor compreensão da classe *cvector* em questão.

Outro ponto de interesse, a saber, é que o princípio de funcionamento desta classe é muito semelhante ao da classe *dvector*, diferindo apenas pela exigência da indicação de um número de inicialização para um vetor a ser construído, mesmo que este seja nulo (zero) e, obviamente, por funções específicas aos números complexos.

4.6.2. - Funções da Classe

4.6.2.1. - Construtores e Destrutor

Construtores/Destrutor	**Propósito**
cvector();	declarar uma variável como do tipo *cvector*
cvector(int size, complex initvalue);	declarar e inicializar uma variável *cvector* com um dado tamanho e com todos elementos iguais
cvector(int size, complex initvalues);*	declarar e inicializar uma variável *cvector* a partir de um *array* (estático ou dinâmico)
cvector(const cvector& argm);	declarar e inicializar um *cvector* a partir de outro e garantir, quando necessária, a conversão adequada de tipos
cvector(const dvector& p_real, const dvector& p_imag);	declarar e inicializar um *cvector* a partir de dois *dvectors* (parte real e imaginária)
cvector(const dvector& argm);	declarar e inicializar um *cvector* a partir de um *dvector* e garantir, quando necessária, a conversão adequada de tipos
virtual ~cvector();	(destrutor da classe)

Quadro 4.9. - Construtores e destrutor da classe *cvector*.

4.6.2.2. - Funções e Operadores de Entrada e Saída de Dados

Funções/Operadores	Propósito
friend istream& operator>> (istream& in, cvector& argm);	propiciar a entrada de dados via *console*
friend ifstream& operator>> (ifstream& fin, cvector& argm);	propiciar a entrada de dados via arquivo ASCII
friend ostream& operator<< (ostream& out, const cvector& argm);	propiciar a saída de dados via *console*
friend ofstream& operator<< (ifstream& fout, const cvector& argm);	propiciar a saída de dados via arquivo ASCII
int size() const;	fornecer o tamanho do *cvector*
complex& operator[] (int index) const;	permitir a leitura e escrita de elemento no vetor
void setval(int index, complex value);	permitir a escrita de elemento no vetor (em posições existentes ou não)

Quadro 4.10. - Funções e operadores de entrada e saída de dados da classe *cvector*.

4.6.2.3. - Demais Funções e Operadores

Funções/Operadores	Propósito
cvector operator= (const cvector& rval);	realizar a atribuição de vetores
cvector operator+ (const cvector& rval) const; *friend cvector operator+ (const cvector& lval, const complex rval);* *friend cvector operator+ (const complex lval, const cvector& rval);*	realizar a soma de vetores ou a soma de vetor com um número complexo (apesar desta última não ser matematicamente definida)
cvector operator- (const cvector& rval) const; *cvector operator- (const complex rval) const;* *cvector operator- () const;*	realizar ou a subtração de vetores, ou a subtração de vetor por um número complexo (apesar de não ser matematicamente definida) ou o unário menos
friend cvector operator (const cvector& lval, const complex rval);* *friend cvector operator* (const complex lval, const cvector& rval);*	realizar a multiplicação de vetor por um número complexo
cvector operator/ (const complex rval) const;	realizar a divisão de vetor por um número complexo

Quadro 4.11. - Demais funções e operadores da classe *cvector*.

Funções/Operadores	Propósito
int operator== (const cvector& rval) const;	propiciar a comparação de igualdade de vetores
int operator!= (const cvector& rval) const;	propiciar a comparação de desigualdade de vetores
cvector conjug() const;	obter o vetor conjugado do vetor de números complexos

Quadro 4.11. - Demais funções e operadores da classe *cvector* (continuação).

4.7. - Classe *cmatrix*

4.7.1. - Sinopse da Classe

Semelhante às demais, a classe *cmatrix* assume a responsabilidade pela abstração de dados do tipo matriz de números complexos (tipo *complex*). Esta foi criada para atender a necessidade de matrizes de números complexos, que necessitem de alocação dinâmica de memória e que exijam recursos de entrada e saída de dados, assim como de uma aritmética básica, contando inclusive com as funções de cálculo de transposta, determinante e inversa.

O conhecimento da utilização de números complexos é de fundamental importância para a compreensão desta classe, logo uma leitura prévia dos manuais, citados na seção 4.6.1., facilitará a assimilação da documentação por parte dos usuários da classe em questão.

A manipulação desta é similar a manipulação dos tipos internos ao compilador C++. O acesso aos elementos dos vetores continua sendo feito pelos operadores [][] (colchetes); o uso dos operadores =, +, -, *, /, == e != também é permitido; além destes operadores, foi definido o operador →* para descrever o produto ponto a ponto de matrizes complexas. A entrada e saída de dados pode ser feita com os operadores >> (entrar em) e << (sair de), tanto por meio do *console* (teclado + monitor) como por meio de arquivos, ou através de funções de acesso aos dados internos (atributos) da classe; e ainda, para dar uma melhor flexibilidade de declaração e inicialização de variáveis, foram concebidos nove construtores de tipo.

4.7.2. - Funções da Classe

4.7.2.1. - Construtores e Destrutor

Construtores/Destrutor	Propósito
cmatrix();	declarar uma variável como do tipo *cmatrix*
cmatrix(int mrows, int mcols);	declarar e inicializar uma variável *cmatrix* com um dado número de linhas, de colunas e com todos elementos iguais a zero
cmatrix(int mrows, int mcols, complex initvalue);	declarar e inicializar uma variável *cmatrix* com um dado número de linhas, de colunas e com todos elementos iguais
cmatrix(int mrows, int mcols, complex initvalues);*	declarar e inicializar uma variável *cmatrix* a partir de um *array* (estático ou dinâmico), o qual deve ser arranjado por linhas
cmatrix(int mrows, cvector initvectors);*	declarar e inicializar uma variável *cmatrix* a partir de um *array* (estático ou dinâmico) de *cvectors* e supondo-os como vetores-linha
cmatrix(char flag, int dimension);*	declarar e inicializar uma variável *cmatrix* como uma matriz identidade
cmatrix(const cmatrix& argm)	declarar e inicializar uma *cmatrix* a partir de outra e garantir, quando necessária, a conversão adequada de tipos
cmatrix(const dmatrix& p_real, const dmatrix& p_imag);	declarar e inicializar uma *cmatrix* a partir de duas matrizes do tipo *dmatrix* (parte real e imaginária)
cmatrix(const dmatrix& argm);	declarar e inicializar uma *cmatrix* a partir de uma *dmatrix* e garantir, quando necessária, a conversão adequada de tipos
virtual ~cmatrix();	(destrutor da classe)

Quadro 4.12. - Construtores e destrutor da classe *cmatrix*.

4.7.2.2. - Funções e Operadores de Entrada e Saída de Dados

Funções/Operadores	Propósito
friend istream& operator>> (istream& in, cmatrix& argm);	propiciar a entrada de dados via *console*
friend ifstream& operator>> (ifstream& fin, cmatrix& argm);	propiciar a entrada de dados via arquivo ASCII

Quadro 4.13. - Funções e operadores de entrada e saída de dados da classe *cmatrix*.

Funções/Operadores	Propósito
friend ostream& operator<< (ostream& out, const cmatrix& argm);	propiciar a saída de dados via *console*
friend ofstream& operator<< (ifstream& fout, const cmatrix& argm);	propiciar a saída de dados via arquivo ASCII
int rows() const;	fornecer o número de linhas da *cmatrix*
int cols() const;	fornecer o número de colunas da *cmatrix*
cvector& operator[] (int index) const;	permitir o acesso a vetores da matriz, e em conjunto com o operador [] da classe *cvector*, permite leitura ou escrita de elemento na matriz
void setval(int rowindex, int colindex, complex value);	permitir a escrita de elemento na matriz (em posições existentes ou não)
void setrow(int rowindex, cvector vvalue);	permitir a escrita, em uma linha existente ou não, de um *cvector* (assumindo-o como vetor-linha) na matriz
void setcol(int colindex, cvector vvalue);	permitir a escrita, em uma coluna existente ou não, de um *cvector* (assumindo-o como vetor-coluna) na matriz
void copyrow(const cmatrix& from, int from_row, int this_row);	copiar uma linha de uma matriz para outra
void copycol(const cmatrix& from, int from_col, int this_col);	copiar uma coluna de uma matriz para outra

Quadro 4.13. - Funções e operadores de entrada e saída de dados da classe *cmatrix* (continuação).

4.7.2.3. - Demais Funções e Operadores

Funções/Operadores	Propósito
void switrow(int row1, int row2);	realizar a simples permuta de linhas de uma matriz
void switcol(int col1, int col2);	realizar a simples permuta de colunas de uma matriz
cmatrix operator= (const cmatrix& rval);	realizar a atribuição de matrizes
cmatrix operator+ (const cmatrix& rval) const; *friend cmatrix operator+ (const cmatrix& lval, const complex rval);* *friend cmatrix operator+ (const complex lval, const cmatrix& rval);*	realizar a soma de matrizes ou a soma de matriz com um número complexo (apesar desta última não ser matematicamente definida)

Quadro 4.14. - Demais funções e operadores da classe *cmatrix*.

Funções/Operadores	Propósito
cmatrix operator- (const cmatrix& rval) const; *cmatrix operator- (const complex rval) const;* *cmatrix operator- () const;*	realizar ou a subtração de matrizes, ou a subtração de matriz por um número complexo (apesar de não ser matematicamente definida) ou o unário menos
cmatrix operator (const cmatrix& rval) const;* *friend cmatrix operator* (const cmatrix& lval, const complex rval);* *friend cmatrix operator* (const complex lval, const cmatrix& rval);*	realizar a multiplicação de matrizes ou de matriz por um número complexo
cmatrix operator→ (const cmatrix& rval) const;*	realizar o produto ponto a ponto de matrizes, isto é, C = A →* B é tal que A e B tem mesmas dimensões e cada elemento da matriz C é dado por $c_{ij} = a_{ij} * b_{ij}$, onde i e j são, respectivamente, os índices de linha e de coluna das matrizes
cmatrix operator/ (const complex rval) const;	realizar a divisão de matriz por um número complexo
int operator== (const cmatrix& rval) const;	propiciar a comparação de igualdade de matrizes
int operator!= (const cmatrix& rval) const;	propiciar a comparação de desigualdade de matrizes
cmatrix conjug() const;	obter a matriz conjugada da matriz de números complexos
complex determinant() const;	obter o determinante da matriz
cmatrix transpose() const;	obter a matriz transposta
cmatrix inverse() const;	obter a matriz inversa

Quadro 4.14. - Demais funções e operadores da classe *cmatrix* (continuação).

4.8. - Classe *extcmat*

4.8.1. - Sinopse da Classe

A classe *extcmat* - *extended cmatrix* - foi desenvolvida como um aprimoramento da classe *cmatrix*. Este aprimoramento foi realizado utilizando-se os recursos de herança da linguagem C++, que por conseguinte, permitiram preservar para esta classe todos os recursos da classe base *cmatrix*.

Dentre os recursos mantidos, estão os nove construtores bases, os operadores de entrada (>>) e saída (<<) de dados, as funções (e operadores) de acesso ou de simples manipulação algébrica dos elementos da matriz (*rows*, *cols*, *operator* [], *setval*, *setrow*, *setcol*, *copyrow*, *copycol*, *switrow* e *switcol*), os operadores matemáticos (=, +, -, *, →*, /, == e !=) e as funções matemáticas básicas (*determinant*, *transpose* e *inverse*).

Para esta classe, foram acrescentados alguns recursos mais específicos à área de sistemas de controle e áreas afins. Ou seja, estão disponíveis também para esta classe as funções *rank* - cálculo do posto, *eigenvalues* - cálculo de autovalores do problema $Ax = \lambda x$ e *eigenvalvec* - cálculo de autovalores e autovetores associados do problema $Ax = \lambda x$. Além destes novos recursos, também aparece um novo construtor responsável pela conversão (silenciosa) do tipo *cmatrix* para o tipo *extcmat*, de modo que o uso do tipo *cmatrix*, quando se trabalha com o tipo *extcmat*, é plenamente possível.

4.8.2. - Funções da Classe

4.8.2.1. - Construtores e Destrutor

Construtores/Destrutor	Propósito
extcmat();	declarar uma variável como do tipo *extcmat*, utilizando um construtor da classe base *cmatrix*
extcmat(int mrows, int mcols);	declarar e inicializar uma variável *extcmat* com um dado número de linhas, de colunas e com todos elementos iguais a zero, utilizando um construtor da classe base *cmatrix*
extcmat(int mrows, int mcols, complex initvalue);	declarar e inicializar uma variável *extcmat* com um dado número de linhas, de colunas e com todos elementos iguais, utilizando um construtor da classe base *cmatrix*
extcmat(int mrows, int mcols, complex initvalues);*	declarar e inicializar uma variável *extcmat*, utilizando um construtor da classe base *cmatrix*, a partir de um *array* (estático ou dinâmico), o qual deve ser arranjado por linhas
extcmat(int mrows, cvector initvectors);*	declarar e inicializar uma variável *extcmat*, utilizando um construtor da classe base *cmatrix*, a partir de um *array* (estático ou dinâmico) de *cvectors* e supondo-os como vetores-linha
extcmat(char flag, int dimension);*	declarar e inicializar uma variável *extcmat* como uma matriz identidade, utilizando um construtor da classe base *cmatrix*
extcmat(const extcmat& argm);	declarar e inicializar uma *extcmat*, utilizando um construtor da classe base *cmatrix*, a partir de outra e garantir, quando necessária, a conversão adequada de tipos

Quadro 4.15. - Construtores e destrutor da classe *extcmat*.

Construtores/Destrutor	Propósito
extcmat(const cmatrix& argm);	declarar e inicializar uma *extcmat* a partir de uma *cmatrix* ou garantir, quando necessária, a conversão silenciosa do tipo *cmatrix* no tipo *extcmat*
extcmat(const dmatrix& p_real, const dmatrix& p_imag);	declarar e inicializar uma *extcmat* a partir de duas matrizes do tipo *dmatrix* (parte real e imaginária), utilizando um construtor da classe base *cmatrix*; note que este garante também a declaração e inicialização de uma *extcmat* a partir de duas matrizes do tipo *extdmat* (parte real e imaginária), já que o tipo *extdmat* é herdeiro do tipo *dmatrix*
extcmat(const dmatrix& argm);	declarar e inicializar uma *extcmat* a partir de uma *dmatrix* e garantir, quando necessária, a conversão adequada de tipos, utilizando para tanto um construtor da classe base *cmatrix*; note que este garante também a declaração e inicialização de uma *extcmat* a partir de uma *extdmat*, já que o tipo *extdmat* é herdeiro do tipo *dmatrix*
virtual ~extcmat();	(destrutor da classe)

Quadro 4.15. - Construtores e destrutor da classe *extcmat* (continuação).

4.8.2.2. - Funções e Operadores de Entrada e Saída de Dados

Similarmente a classe *dmatrix*, esta classe herda todas as funções e operadores de entrada e saída de dados já disponíveis na classe base *cmatrix*. Para conhecer estas, basta consultar o quadro 4.13. deste capítulo.

4.8.2.3. - Demais Funções e Operadores

Antes de situar as outras funções e operadores específicas para esta classe, ressalta-se que as descritas no quadro 4.14. são igualmente válidas para esta. Segue então, o conjunto restante dos métodos desta classe:

Funções/Operadores	Propósito
int rank(complex tol = 0.) const;	calcular o posto da matriz
int eigenvalues(extcmat& eigval) const;	calcular os autovalores da matriz no problema Ax = λx
int eigenvalvec(extcmat& eigval, extcmat& eigvec) const;	calcular os autovalores e autovetores associados da matriz no problema Ax = λx

Quadro 4.16. - Demais funções e operadores da classe *extcmat*.

4.9. - Classe *polynom*

4.9.1. - Sinopse da Classe

A classe *polynom* é a responsável pela abstração de dados do tipo polinômio de ordem positiva ou nula e com coeficientes reais. Este tipo foi desenvolvido para propiciar um conjunto de facilidades quando da manipulação de polinômios. Para tanto, foram implementados recursos de entrada e saída de dados - os operadores >> (entrar em) e << (sair de) e as funções de acesso aos dados internos (atributos) da classe - bem como recursos matemáticos - os operadores aritméticos (=, +, -, *, /, %, == e !=), as funções de teste de polinômio nulo (função *iszero*), de cálculo de raízes (função *roots*), de cálculo do valor do polinômio num dado ponto (função *polyval*), de obtenção da derivada indefinida (função *derivative*) e da integral indefinida (função *integral*)

Como nas demais classes implementadas, a classe *polynom* ostenta flexibilidade na declaração e inicialização de dados, ou seja, existe mais de uma maneira de declarar e inicializar uma variável *polynom*. Enfim, o usuário da classe *polynom* notará que o manuseio de polinômios é bastante simples e eficiente, pois pode-se trabalhar com polinômios de forma semelhante àquela relativa aos tipos internos da linguagem C++. E além disso, toda esta estrutura funciona consoante alocação dinâmica de memória, de modo que o consumo de memória disponível é otimizada.

4.9.2. - Funções da Classe

4.9.2.1. - Construtores e Destrutor

Construtores/Destrutor	Propósito
polynom();	declarar uma variável como do tipo *polynom*
polynom(double initvalue);	declarar um escalar como um polinônio de ordem zero
polynom(int order, double initvalue);	declarar e inicializar um polinômio com todos coeficientes iguais
polynom(int order, double initvalues);*	declarar e inicializar uma variável *polynom* a partir de um *array* (estático ou dinâmico)
polynom(const polynom& argm);	declarar e inicializar um *polynom* a partir de outro e garantir, quando necessária, a conversão adequada de tipos
virtual ~polynom();	(destrutor da classe)

Quadro 4.17. - Construtores e destrutor da classe *polynom*.

4.9.2.2. - Funções e Operadores de Entrada e Saída de Dados

Funções/Operadores	Propósito
friend istream& operator>> (istream& in, polynom& argm);	propiciar a entrada de dados via *console*
friend ifstream& operator>> (ifstream& fin, polynom& argm);	propiciar a entrada de dados via arquivo ASCII
friend ostream& operator<< (ostream& out, const polynom& argm);	propiciar a saída de dados via *console*
friend ofstream& operator<< (ifstream& fout, const polynom& argm);	propiciar a saída de dados via arquivo ASCII
int order() const;	fornecer a ordem do *polynom*
double getval(int index) const;	permitir a leitura de coeficiente do polinômio
void setval(int index, double value);	permitir a escrita de coeficiente em qualquer termo do polinômio

Quadro 4.18. - Funções e operadores de entrada e saída de dados da classe *polynom*.

4.9.2.3. - Demais Funções e Operadores

Funções/Operadores	Propósito
polynom operator= (const polynom& rval);	realizar a atribuição de polinômios
friend polynom operator+ (const polynom& lval, const polynom& rval);	realizar a adição de polinômios
friend polynom operator- (const polynom& lval, const polynom& rval); *polynom operator- () const;*	realizar a subtração de polinômios ou o unário menos
friend polynom operator (const polynom& lval, const polynom& rval);*	realizar a multiplicação de polinômios
friend polynom operator/ (const polynom& lval, const polynom& rval);	realizar a divisão de polinômios
friend polynom operator% (const polynom& lval, const polynom& rval);	realizar a operação resto (ou módulo) da divisão de polinômios
friend int operator== (const polynom& lval, const polynom& rval);	propiciar a comparação de igualdade de polinômios
friend int operator!= (const polynom& lval, const polynom& rval);	propiciar a comparação de desigualdade de polinômios
int iszero() const;	testar se o polinômio é uma constante nula
complex polyval(complex value) const;	calcular o valor (complexo) do polinômio num dado ponto (complexo)
cvector roots() const;	calcular as raízes (complexas) do polinômio
polynom derivative() const;	obter a derivada indefinida do polinômio
polynom integral(double constant = 0.) const;	obter a integral indefinida do polinômio

Quadro 4.19. - Demais funções e operadores da classe *polynom*.

4.10. - Classe *trfunct*

4.10.1. - Sinopse da Classe

A classe *trfunct* é a responsável pela abstração de dados do tipo função de transferência linear, causal e invariante no tempo, definida sob condições iniciais nulas. Este tipo foi desenvolvido para propiciar um conjunto de facilidades quando da manipulação de funções de transferência (F.T.). Para tanto, foram implementados recursos de entrada e saída de dados

(operadores >> e <<, respectivamente, e funções de acesso aos dados), recursos matemáticos, como os operadores aritméticos (=, +, -, *, == e !=) ou a função *closeloop*, e ainda recursos para obtenção da resposta temporal, da resposta freqüencial ou do lugar geométrico das raízes.

A classe *trfunct* apresenta flexibilidade na declaração e inicialização de dados, porquanto deixa disponível diversos modos de declarar e inicializar uma variável *trfunct*. Sucintamente, pode-se entrar dados tanto no formato polinomial como no formato fatorado, como mostrado a seguir:

- Forma Polinomial

$$F.T.(s) = \frac{b_m s^m + b_{m-1} s^{m-1} + \ldots + b_0}{a_n s^n + a_{n-1} s^{n-1} + \ldots + a_0} + d$$

onde: b_m, b_{m-1}, ..., b_o são os coeficientes reais do numerador ($m \geq 0$),
a_n, a_{n-1}, ..., a_o são os coeficientes reais do denominador ($n \geq m$), e
d é o escalar de transmissão direta da função de transferência.

- Forma Fatorada

$$F.T.(s) = g \cdot \frac{(s - z_0)(s - z_1)\ldots(s - z_{nz})}{(s - p_0)(s - p_1)\ldots(s - p_{np})}$$

onde: g é o ganho da função de transferência,
z_o, z_1, ..., z_{nz} são os zeros (complexos) do numerador ($nz \geq 0$, se existir), e
p_o, p_1, ..., p_{np} são os pólos (complexos) do denominador ($np \geq nz$, se existir).

Do ponto de vista do usuário, esta classe caracteriza-se por possuir tratamento simples de dados, já que segue basicamente o comportamento de tipos internos da linguagem C++. A princípio, o usuário não necessita conhecer a estrutura de dados internos da classe, mas é conveniente mencionar que a classe opera internamente com duas representações - a polinomial mônica (estritamente própria) e a fatorada padrão (própria); de fato, isto foi feito com finalidade de garantir uma maior eficiência para a obtenção da resposta temporal e da resposta freqüencial. Além disso, esta classe conta também com o uso do princípio de alocação dinâmica de memória, provendo desta forma uma constante otimização de memória disponível.

Convém por fim salientar que, para criar facilidades computacionais em relação às funções de transferência acima definidas, também entender-se-á um número real puro (uma constante) como possível F.T.

4.10.2. - Funções da Classe

4.10.2.1. - Construtores e Destrutor

Construtores/Destrutor	Propósito
trfunct();	declarar uma variável como do tipo *trfunct*
trfunct(double gain);	declarar um escalar como uma função de transferência
trfunct(int m, double num, int n, double* den, double d = 0.);*	declarar e inicializar uma função de transferência a partir de dois *arrays* (estáticos ou dinâmicos), contendo os coeficientes do numerador e do denominador, e de um escalar de transmissão direta
trfunct(const polynom& num, const polynom& den, double d = 0.);	declarar e inicializar uma variável *trfunct* a partir de dois polinômios - numerador e denominador - e de um escalar de transmissão direta
trfunct(double gain, int nz, complex zero, int np, complex* pole);*	declarar e inicializar uma função de transferência a partir de um escalar (ganho da F.T.) e de dois *arrays* (estáticos ou dinâmicos) de complexos, contendo os zeros e pólos da função de transferência
trfunct(double gain, const cvector& zero, const cvector& pole);	declarar e inicializar uma variável *trfunct* a partir de um escalar (ganho da F.T.) e de dois vetores complexos com os zeros e pólos da função de transferência
trfunct(const trfunct& argm);	declarar e inicializar uma *trfunct* a partir de outra e garantir, quando necessária, a conversão adequada de tipos
virtual ~trfunct();	(destrutor da classe)

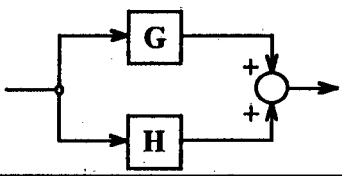
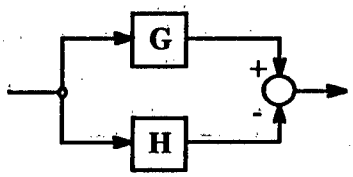
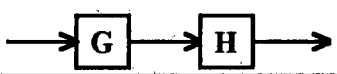
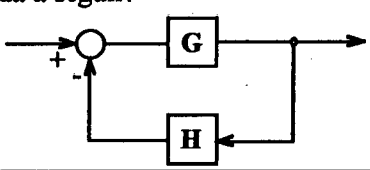
Quadro 4.20. - Construtores e destrutor da classe *trfunct*.

4.10.2.2. - Funções e Operadores de Entrada e Saída de Dados

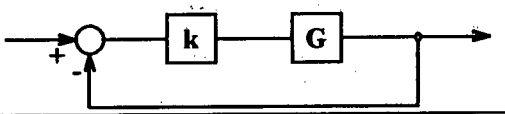
Funções/Operadores	Propósito
friend istream& operator>> (istream& in, trfunct& argm);	propiciar a entrada de dados via *console*
friend ifstream& operator>> (ifstream& fin, trfunct& argm);	propiciar a entrada de dados via arquivo ASCII
friend ostream& operator<< (ostream& out, const trfunct& argm);	propiciar a saída de dados via *console*
friend ofstream& operator<< (ifstream& fout, const trfunct& argm);	propiciar a saída de dados via arquivo ASCII
polynom getnum() const;	obter o numerador da F.T.
polynom getden() const;	obter o denominador da F.T.
double getdtr() const;	obter o escalar de transmissão direta da F.T.
int ordnum() const;	obter o grau do numerador da F.T. (forma estritamente própria)
int ordden() const;	obter o grau do denominador da F.T. (forma estritamente própria)
double cffnum(int index) const;	obter um coeficiente do numerador da F.T. (forma estritamente própria)
double cffnum(int index) const;	obter um coeficiente do denominador da F.T. (forma estritamente própria)
double getgtf() const;	obter o ganho da F.T.
cvector getzer() const;	obter o(s) zero(s) da F.T.
cvector getpol() const;	obter o(s) pólo(s) da F.T.
int qttzer() const;	obter o número de zeros da F.T.
int qttpol() const;	obter o número de pólos da F.T.
complex onezer(int index) const;	obter um zero da F.T.
complex onepol(int index) const;	obter um pólo da F.T.

Quadro 4.21. - Funções e operadores de entrada e saída de dados da classe *trfunct*.

4.10.2.3. - Demais Funções e Operadores

Funções/Operadores	Propósito
trfunct operator= (const trfunct& rval);	realizar a atribuição de funções de transferência
friend trfunct operator+ (const trfunct& lval, const trfunct& rval);	realizar a adição de funções de transferência [G(s) + H(s)], ou seja, obter a função de transferência resultante de duas outras em paralelo, conforme mostra o seguinte modelo: 
friend trfunct operator- (const trfunct& lval, const trfunct& rval); *trfunct operator- () const;*	realizar a subtração de funções de transferência [G(s) - H(s)], ou seja, obter a função de transferência resultante de duas outras em paralelo, conforme mostra o seguinte modelo:  ou ainda o unário menos, conforme mostra a figura a seguir: 
friend trfunct operator (const trfunct& lval, const trfunct& rval);*	realizar a multiplicação de funções de transferência [G(s) * H(s)], ou seja, obter a função de transferência resultante de duas outras em cascata, segundo mostra o seguinte modelo: 
trfunct closeloop(const trfunct& h) const;	obter a função de transferência resultante do modelo em malha fechada a seguir: 
friend int operator== (const trfunct& lval, const trfunct& rval);	propiciar a comparação de igualdade de funções de transferência
friend int operator!= (const trfunct& lval, const trfunct& rval);	propiciar a comparação de desigualdade de funções de transferência
int iszero() const;	testar se a função de transferência é uma constante nula

Quadro 4.22. - Demais funções e operadores da classe *trfunct*.

Funções/Operadores	Propósito
double timeresp(double step, double u0, double u1, double u2, dvector& x) const;	obter um dado instante da resposta temporal de uma F.T. a partir de um dado sinal de entrada e de uma dinâmica preestabelecida
double modulus(double w) const;	obter a resposta freqüencial em termos do módulo (ou ganho) da F.T. para uma dada frequência
double mod_dB(double w) const;	obter a resposta freqüencial em termos do módulo (ou ganho), já dado em decibeis (dB), da F.T. para uma dada freqüência
double phase(double w) const;	obter a resposta freqüencial em termos da fase da F.T., dada em graus, para uma dada frequência
complex real_imag(double w) const;	obter a resposta freqüencial em termos do módulo e fase da F.T., representados pelas partes real e imaginária, para uma dada frequência
cvector rootlocus(double k) const;	obter o lugar geométrico das raízes (LGR) para um dado ganho de malha direta 

Quadro 4.22. - Demais funções e operadores da classe *trfunct* (continuação).

4.11. - Classe *statvar*

4.11.1. - Sinopse da Classe

A classe *statvar* é a responsável pela abstração de dados do tipo sistemas de equações de estado lineares e invariantes no tempo. Este tipo foi desenvolvido para propiciar um conjunto de facilidades quando da manipulação dos sistemas de equações de estado (S.E.E.), descritos na forma:

$$\begin{cases} \dot{x} = Ax + Bu \\ y = Cx + Du \end{cases}$$

onde u é um vetor de entrada,
x é um vetor de estados,
y é um vetor de saída,
A, B, C e D são matrizes contendo a dinâmica do sistema.

Para tanto, foram implementados recursos de entrada e saída de dados (operadores >> e <<, respectivamente, e funções de acesso aos dados), recursos matemáticos, como os operadores aritméticos (=, +, -, *, == e !=), e ainda recursos para obtenção da resposta temporal, da matriz de realimentação estática de estados, ou então, do sistema com realimentação de estática de estados.

A classe *statvar* apresenta flexibilidade na declaração e inicialização de dados, porquanto deixa disponível mais de um modo de declarar e inicializar uma variável do tipo *statvar*. Do ponto de vista do usuário, esta classe caracteriza-se também por possuir tratamento simples de dados, já que segue basicamente o comportamento de tipos internos da linguagem C++.

O usuário não necessita conhecer a estrutura dos dados internos da classe, mas é conveniente mencionar que a classe opera internamente fazendo uso de matrizes do tipo *extdmat* que, por sua vez, propicia desde recursos de manipulação de matrizes até, por exemplo, o cálculo de posto ou de autovalores. Além disso, esta classe conta inclusive com o uso do princípio de alocação dinâmica de memória, provendo desta forma uma constante otimização de memória disponível.

4.11.2. - Funções da Classe

4.11.2.1. - Construtores e Destrutor

Construtores/Destrutor	**Propósito**
statvar();	declarar uma variável como do tipo *statvar*
statvar(int inputs, int states, int outputs, double array_A, double* array_B, double* array_C, double* array_D);*	declarar e inicializar um sistema de equações de estado a partir de quatro *arrays* (estáticos ou dinâmicos), descrevendo as matrizes A, B, C e D do S.E.E.
statvar(extdmat matrix_A, extdmat matrix_B, extdmat matrix_C, extdmat matrix_D);	declarar e inicializar um sistema de equações de estado a partir de quatro matrizes do tipo *extdmat*, descrevendo as matrizes A, B, C e D do S.E.E.
statvar(const statvar& argm);	declarar e inicializar uma *statvar* a partir de outra e garantir, quando necessária, a conversão adequada de tipos
virtual ~statvar();	(destrutor da classe)

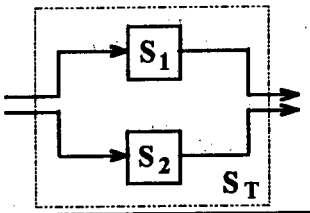
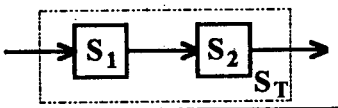
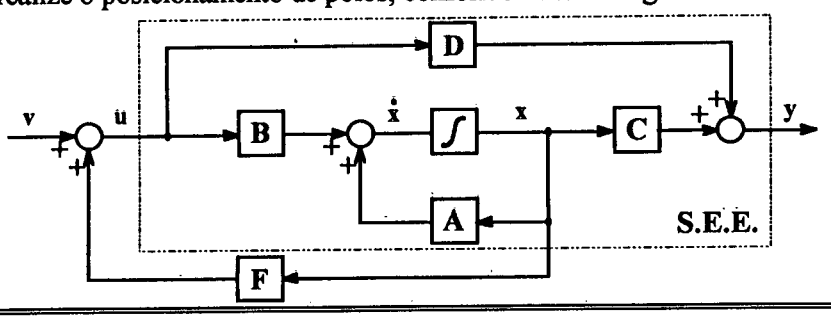
Quadro 4.23. - Construtores e destrutor da classe *statvar*.

4.11.2.2. - Funções e Operadores de Entrada e Saída de Dados

Funções/Operadores	Propósito
friend istream& operator>> (istream& in, statvar& argm);	propiciar a entrada de dados via *console*
friend ifstream& operator>> (ifstream& fin, statvar& argm);	propiciar a entrada de dados via arquivo ASCII
friend ostream& operator<< (ostream& out, const statvar& argm);	propiciar a saída de dados via *console*
friend ofstream& operator<< (ifstream& fout, const statvar& argm);	propiciar a saída de dados via arquivo ASCII
int ninputs() const;	obter o número de entradas do S.E.E.
int nstates() const;	obter o número de estados do S.E.E.
int noutputs() const;	obter o número de saídas do S.E.E.
extdmat getA() const;	obter a matriz A do S.E.E.
extdmat getB() const;	obter a matriz B do S.E.E.
extdmat getC() const;	obter a matriz C do S.E.E.
extdmat getD() const;	obter a matriz D do S.E.E.
double getvalA(int rowindex, int colindex) const;	obter um dado elemento da matriz A do S.E.E.
double getvalB(int rowindex, int colindex) const;	obter um dado elemento da matriz B do S.E.E.
double getvalC(int rowindex, int colindex) const;	obter um dado elemento da matriz C do S.E.E.
double getvalD(int rowindex, int colindex) const;	obter um dado elemento da matriz D do S.E.E.
double setvalA(int rowindex, int colindex, double value) const;	escrever um dado elemento na matriz A do S.E.E.
double setvalB(int rowindex, int colindex, double value) const;	escrever um dado elemento na matriz B do S.E.E.
double setvalC(int rowindex, int colindex, double value) const;	escrever um dado elemento na matriz C do S.E.E.
double setvalD(int rowindex, int colindex, double value) const;	escrever um dado elemento na matriz D do S.E.E.

Quadro 4.24. - Funções e operadores de entrada e saída de dados da classe *statvar*.

4.11.2.3. - Demais Funções e Operadores

Funções/Operadores	Propósito
statvar operator= (const statvar& rval);	realizar a atribuição de sistemas de equações de estado
friend statvar operator+ (const statvar& lval, const statvar& rval);	realizar a adição de sistemas de equações de estado [$S_T = S_1 + S_2$], ou seja, obter o sistema de equação de estado resultante de dois outros em paralelo, consoante mostra o seguinte modelo: 
friend statvar operator (const statvar& lval, const statvar& rval);*	realizar a multiplicação de sistemas de equações de estado [$S_T = S_1 * S_2$], ou seja, obter o sistema de equação de estado resultante de dois outros em cascata, consoante mostra o seguinte modelo: 
friend int operator== (const statvar& lval, const statvar& rval);	propiciar a comparação de igualdade de sistemas de equações de estado
friend int operator!= (const statvar& lval, const statvar& rval);	propiciar a comparação de desigualdade de sistemas de equações de estado
dvector timeresp(double step, dvector& u0, dvector& u1, dvector& u2, dvector& x) const;	obter um dado instante da resposta temporal de um S.E.E. a partir de um dado sinal de entrada e de uma dinâmica preestabelecida primeiramente apresentada como condições iniciais
extdmat poleplacement(cvector& poles, double ldtol = 0., double ortol = 1., int maxit = 5) const;	obter a matriz F de realimentação estática de estados do S.E.E. que realize o posicionamento de pólos, conforme ilustra a figura abaixo: 

Quadro 4.25. - Demais funções e operadores da classe *statvar*.

Funções/Operadores	Propósito
statvar staticstatefb(const extdmat& F) const;	obter o S.E.E. com a matriz F(m x n) de realimentação estática de estados, dado por: $\begin{cases} \dot{x} = (A + BF)x + Bv \\ y = (C + DF)x + Dv \end{cases}$ onde *v* é um vetor de entrada, *x* é um vetor de estados, *y* é um vetor de saída, *F* é a matriz de realimentação estática de estados, *(A+BF)* é a nova matriz dinâmica do S.E.E., *B* é a matriz de entrada do S.E.E., *(C+DF)* é a nova matriz de saída do S.E.E., e *D* é a matriz de transmissão direta do S.E.E.

Quadro 4.25. - Demais funções e operadores da classe *statvar* (continuação).

4.12. - Alguns Resultados Numéricos

Todo o material implementado passou por ensaios, referentes a verificação e validação do mesmo. Nesse sentido, para cada uma das classes anteriormente descritas, existe um respectivo arquivo de teste, responsável pelo chamado controle de qualidade ou homologação das classes. Estes arquivos de teste, inclusive, poderão servir de auxílio para uma continuidade do desenvolvimento, assim como para facilitar as eventuais necessidades de manutenção.

A fim de ilustrar alguns dos testes realizados e demonstrar um pouco do potencial do material implementado, segue um conjunto de exemplos ensaiados com base nos recursos da biblioteca em questão.

4.12.1. - Exemplo de Manipulação de uma Matriz

Seja a matriz A, dada por:

$$A = \begin{bmatrix} 3 & -5 & 6 & 4 & -2 & -3 & 8 \\ 1 & 1 & -9 & 15 & 1 & -9 & 2 \\ 2 & -1 & 7 & 5 & -1 & 6 & 11 \\ -1 & 1 & 3 & 2 & 7 & -1 & -2 \\ 4 & 3 & 1 & -7 & 2 & 1 & 1 \\ 2 & 9 & -8 & 11 & -1 & -4 & -1 \\ 7 & 2 & -1 & 2 & 7 & -1 & 9 \end{bmatrix}$$

Então, com o uso da classe *dmatrix* ou *extdmat*, foi possível obter, por exemplo, a matriz inversa A^{-1}, descrita por:

$$A^{-1} = \begin{bmatrix} 0.812 & -2.09 & -1.29 & -0.37 & -2.78 & 1.13 & 1.67 \\ -0.365 & 0.984 & 0.641 & 0.208 & 1.46 & -0.453 & -0.844 \\ 0.0948 & -0.0531 & 0.00198 & 0.0868 & 0.0311 & 0.0507 & -0.0535 \\ 0.27 & -0.717 & -0.413 & -0.0904 & -1.04 & 0.413 & 0.566 \\ -0.0587 & 0.0404 & 0.0121 & 0.0741 & 0.00741 & -0.0549 & 0.038 \\ 0.282 & -1.06 & -0.566 & -0.205 & -1.48 & 0.537 & 0.855 \\ -0.523 & 1.41 & 0.88 & 0.19 & 1.9 & -0.766 & -1.07 \end{bmatrix}$$

A verificação deste resultado foi realizada com os próprios recursos aritméticos implementados, isto é, realizou-se simplesmente A * A^{-1}, na expectativa de obter uma matriz próxima a matriz identidade. De fato, obteve-se:

$$A * A^{-1} = \begin{bmatrix} 1 & 0 & 3e-15 & 0 & 0 & 0 & 2e-15 \\ 3e-15 & 1 & 2e-15 & -1e-15 & 6e-15 & -4e-16 & -9e-16 \\ 0 & 2e-15 & 1 & 1e-15 & 4e-15 & 0 & 0 \\ 4e-16 & -4e-16 & -9e-16 & 1 & 9e-16 & 0 & 9e-16 \\ 2e-16 & -1e-15 & 1e-15 & 1e-15 & 1 & 3e-16 & -1e-15 \\ 1e-15 & 1e-15 & 7e-16 & -3e-16 & 1e-15 & 1 & -4e-16 \\ 3e-15 & 0 & 2e-15 & 1e-15 & -4e-15 & 9e-16 & 1 \end{bmatrix} \cong I$$

Resultados como os valores do determinante, do elemento máximo, do elemento mínimo, da média aritmética e da variância populacional dos elementos foram obtidos,

outrossim, por funções simples, onde os números resultantes foram, respectivamente: 332570, -9, 15, 1.816327 e 25.70096.

Um outro resultado, obtido da matriz A(7 x 7) em questão, ajuda a ilustrar esta seção e é justamente o cálculo dos autovalores e autovetores da mesma. Os autovalores λ_i, i = 1, 2, ..., 7, calculados foram, respectivamente: 17.90454, -1.243308 + 11.4376j, -1.243308 - 11.4376j, 1.326109 + 0.7869587j, 1.326109 - 0.7869587j, 0.9649299 + 7.62126j e 0.9649299 - 7.62126j, onde considera-se $j = \sqrt{-1}$. Por outro lado, os autovetores associados são dados pela matriz:

$$X = \begin{bmatrix} 0.6271906 & -0.7500452 & -0.5066638 & 2.228004 & 2.159741 & -2.649076 & 3.886398 \\ -0.04392455 & -0.9814964 & 0.3548566 & -0.9120864 & -0.6171001 & 0.9066549 & 0.8999103 \\ 0.6399184 & 0.4507254 & -0.2762126 & 0.2621528 & 0.9154221 & -3.048382 & -1.352186 \\ 0.09263781 & 0.009365144 & -0.2874798 & 0.768073 & 0.9217943 & -0.5985939 & -0.4070013 \\ 0.1745046 & 0.314412 & 0.4047489 & -0.1533619 & -0.3928722 & 2.333529 & -0.1043758 \\ -0.1828669 & -0.2430871 & 1.192045 & 0.8864016 & 0.5510067 & 3.205506 & -0.06645287 \\ 0.5898387 & 0.153495 & 0.3172899 & -1.535278 & -1.65747 & 1.909166 & -1.783228 \end{bmatrix}$$

Tomando, por exemplo, o autovalor real λ_1 = 17.90454 e seu respectivo autovetor associado x_1 = [0.6271906 -0.04392455 0.639399184 0.09263781 -0.1828669 0.05898387]^t e, ainda conhecendo a relação Ax = λx, pôde-se realizar, através dos próprios recursos implementados, um simples cálculo auxiliar de teste, mostrado a seguir:

$$Ax_1 - \lambda x_1 = \begin{bmatrix} 0 \\ -4.9e-15 \\ 5.3e-15 \\ 5.6e-15 \\ 3.1e-15 \\ -7.1e-15 \\ 2.1e-14 \end{bmatrix}$$

Enfim, estas são algumas das funcionalidades oferecidas pela biblioteca. Um resumo do potencial da biblioteca, em termos de manipulação de vetores e matrizes, já foi devidamente apresentado nas seções de 4.3. à 4.8. e, se necessário, uma documentação completa

pode ser encontrada nos "Manuais de Utilização de Classes C++"; com código de catalogação[1] de CL-0002 a CL-0007.

4.12.2. - Exemplo de Obtenção das Raízes de um Polinômio

Considere o polinômio dado por:

$$P(s) = s^8 + 2s^7 + 3s^6 + 4s^5 + 5s^4 + 6s^3 + 7s^2 + 8s + 9$$

A seguir, aparece um quadro comparativo entre as raízes do polinômio P(s) calculadas pela classe *polynom* e pelo MATLAB®[2]. Observe que os resultados são os mesmos, apenas diferindo no formato, pois o exemplo realizado com classe *polynom* retornou números com 4 dígitos significativos e o MATLAB® números de ponto fixo com 4 casas decimais. Esta comparação com o MATLAB® foi feita, à título de exemplificação, para elucidar a precisão numérica.

Classe *polynom*	MATLAB®
-1.289 - 0.4477j	-1.2888 - 0.4477j
-1.289 + 0.4477j	-1.2888 + 0.4477j
-0.7244 - 1.137j	-0.7244 - 1.1370j
-0.7244 + 1.137j	-0.7244 + 1.1370j
0.1364 - 1.305j	0.1364 - 1.3050j
0.1364 + 1.305j	0.1364 + 1.3050j
0.8767 - 0.8814j	0.8767 - 0.8814j
0.8767 + 0.8814j	0.8767 + 0.8814j

Quadro 4.26. - Comparativo entre as raízes de P(s) calculadas com a classe *polynom* e o MATLAB®.

[1] Conforme estabelecido na norma interna de documentação de classes C++ (ver anexo I), sugerida ao LCMI, as classes implementadas e testadas receberão um código de catalogação na forma CL-*xxxx*.

[2] MATLAB® é um *software* de alto desempenho para a computação numérica e é também marca registrada de The MathWorks, Inc (Referência: [MathWorks 91]).

4.12.3. - Exemplo de Resposta Temporal de uma Função de Transferência

Considere a seguinte função de transferência:

$$G_1(s) = \frac{1176}{s^2 + 48.5s + 1176}$$

O quadro 4.27 apresenta a resposta temporal de $G_1(s)$, calculada com o auxílio da classe *trfunct,* a uma entrada degrau unitário. Estes resultados foram conferidos com o MATLAB® e, respeitada a precisão numérica descrita neste quadro, a comparação resultou em números exatamente iguais. Convém apenas salientar que o passo de integração optado para o cálculo com a classe *trfunct* foi de 0.005 (método de Runge-Kutta de 4ª. Ordem).

Tempo	Classe *trfunct*
0.00	0.0000
0.01	0.0499
0.02	0.1682
0.03	0.3179
0.04	0.4730
0.05	0.6171
0.06	0.7411
0.07	0.8414
0.08	0.9178
0.09	0.9724
0.10	1.0086
0.11	1.0300
0.12	1.0405
0.13	1.0432
0.14	1.0409
0.15	1.0357
0.16	1.0292
0.17	1.0225
0.18	1.0163
0.19	1.0110
0.20	1.0067

Quadro 4.27. - Resposta temporal de $G_1(s)$ calculada com a classe *trfunct*.

Tempo	Classe *trfunct*
0.21	1.0034
0.22	1.0011
0.23	0.9996
0.24	0.9987

Quadro 4.27. - Resposta temporal de $G_1(s)$ calculada com a classe *trfunct* (continuação).

4.12.4. - Exemplo de Resposta Freqüencial de uma Função de Transferência

Um outro exemplo clássico é o de obtenção da resposta freqüencial de uma função de transferência. Então, considere a função de transferência descrita a seguir:

$$G_2(s) = \frac{10(s+10)}{s(s+2)(s+5)}$$

O cálculo da resposta freqüencial, com o uso da classe *trfunct*, possibilitou os resultados descritos no quadro 4.28. dado abaixo:

Logaritmo da Frequência	Freqüência (rad/s)	Módulo (dB)	Módulo	Fase (graus)	Parte Real	Parte Imaginária
-1.000	0.10	40.0	99.86	-93.4	-5.984	-99.681
-0.833	0.15	36.6	67.92	-95.0	-5.965	-67.662
-0.667	0.22	33.3	46.12	-97.4	-5.925	-45.735
-0.500	0.32	29.9	31.19	-100.8	-5.840	-30.636
-0.333	0.46	26.4	20.92	-105.7	-5.665	-20.138
-0.167	0.68	22.8	13.80	-112.7	-5.319	-12.732
-0.000	1.00	18.9	8.81	-122.2	-4.692	-7.462
0.167	1.47	14.5	5.33	-134.3	-3.719	-3.813
0.333	2.15	9.4	2.97	-148.3	-2.524	-1.560
0.500	3.16	3.5	1.50	-162.5	-1.429	-0.452

Quadro 4.28. - Resposta freqüencial de $G_2(s)$ calculada com a classe *trfunct*.

Logaritmo da Freqüência	Freqüência (rad/s)	Módulo (dB)	Módulo	Fase (graus)	Parte Real	Parte Imaginária
0.667	4.64	-3.2	0.69	-174.7	-0.686	-0.064
0.833	6.81	-10.6	0.30	-183.1	-0.296	0.016
1.000	10.09	-18.1	0.12	-187.1	-0.123	0.015
1.167	14.68	-25.6	0.05	-187.7	-0.052	0.007
1.333	21.54	-32.8	0.02	-186.5	-0.023	0.003
1.500	31.62	-39.7	0.01	-184.9	-0.010	0.001
1.667	46.42	-46.5	0.00	-183.5	-0.005	0.000
1.833	68.13	-53.3	0.00	-182.5	-0.002	0.000
2.000	100.00	-60.0	0.00	-181.7	-0.001	0.000
2.167	146.78	-66.7	0.00	-181.2	0.000	0.000
2.333	215.44	-73.3	0.00	-180.8	0.000	0.000
2.500	316.23	-80.0	0.00	-180.5	0.000	0.000
2.667	464.16	-86.7	0.00	-180.4	0.000	0.000
2.833	681.29	-93.3	0.00	-180.3	0.000	0.000
3.000	1000.09	-100.0	0.00	-180.2	0.000	0.000

Quadro 4.28. - Resposta freqüencial de $G_2(s)$ calculada com a classe *trfunct* (continuação).

Para um efeito comparativo, foram calculados através do MATLAB®, o módulo e a fase da função de transferência $G_2(s)$ e, respeitadas a precisão numérica do quadro 4.28., os resultados foram exatamente os mesmos.

4.12.5. - Exemplo de Resposta Temporal de um Sistema de Equações de Estado

Um outro exemplo de resposta temporal, aparece aqui denotando a resposta temporal de um sistema de equações de estado linear e invariante no tempo, descrito na forma a seguir:

$$\begin{cases} \dot{x} = Ax + Bu \\ y = Cx + Du \end{cases}$$

onde u é um vetor de entrada,
x é um vetor de estados,

y é um vetor de saída,

A, B, C e D são, respectivamente, as matrizes dinâmica, de entrada, de saída e de transmissão direta do sistema.

A matrizes A, B, C e D são dadas respectivamente por:

$$A = \begin{bmatrix} -0.0002 & 0.0010 & 0.0054 & 0.0044 & 0.011 & 0 & 0 & -0.040 \\ -0.0012 & -0.0009 & -0.0060 & -0.0019 & -0.002 & 0.003 & 0.047 & 0 \\ 0 & 0 & -0.2500 & 0 & 0 & 0 & 0 & 0 \\ 0.0376 & -0.0667 & 0.0016 & -1.7001 & -0.002 & 0 & -1.444 & -0.004 \\ 0.0311 & 0.0304 & 0.1107 & -0.0106 & -1.557 & 0 & -0.016 & -1.212 \\ 0.0752 & 0.0844 & 0.0203 & 0.0175 & 0.013 & -0.999 & 0.019 & 0.034 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix},$$

$$B = \begin{bmatrix} -0.0062 & 0 & -0.0098 & 0 \\ -0.0173 & 0.0033 & -0.0001 & 0.0028 \\ 0.2486 & 0.0002 & -0.0003 & 0.0002 \\ 0 & 1.4416 & 0 & 0 \\ -0.0001 & 0 & 1.2099 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \quad C = \bar{I}(8x8) \quad \text{e} \quad D = \bar{O}(8x4),$$

com $\bar{I}$ representando a matriz identidade e $\bar{O}$ a matriz nula. Ainda, consider-se-ão as entradas dadas por $u = [\omega_c \; \phi_c \; \theta_c \; r_c]$ e as saídas por $y = [\mu \; v \; \omega \; p \; q \; r \; \phi \; \theta]$.

Uma simulação do sistema, em termos da sua resposta temporal a uma entrada degrau unitário em θ_c ($u = [0 \; 0 \; 1 \; 0]$), é apresentada no quadro 4.29. abaixo:

t	μ	v	ω	p	q	r	ϕ	θ
0	0	0	0	0	0	0	0	0
1	-0.0114	-0.0008	-0.0003	-0.0008	0.5009	0.0065	-0.0003	0.3501
2	-0.0397	-0.0017	-0.0005	-0.0015	0.3272	0.0178	-0.0015	0.7847
3	-0.0835	-0.0022	-0.0006	-0.0016	0.1078	0.0248	-0.0031	0.9945

Quadro 4.29. - Resposta temporal calculada com a classe *statvar*.

t	μ	v	ω	p	q	r	ϕ	θ
4	-0.1338	-0.0024	-0.0008	-0.0015	0.0008	0.0260	-0.0047	1.0395
5	-0.1851	-0.0024	-0.0009	-0.0013	-0.0229	0.0236	-0.0061	1.0240
6	-0.2357	-0.0025	-0.0009	-0.0013	-0.0159	0.0198	-0.0073	1.0036
7	-0.2854	-0.0025	-0.0010	-0.0013	-0.0063	0.0158	-0.0086	0.9929
8	-0.3348	-0.0026	-0.0010	-0.0013	-0.0014	0.0119	-0.0098	0.9894
9	-0.3842	-0.0028	-0.0011	-0.0013	-0.0003	0.0080	-0.0111	0.9887
10	-0.4334	-0.0029	-0.0011	-0.0013	-0.0006	0.0043	-0.0124	0.9883
11	-0.4827	-0.0030	-0.0011	-0.0013	-0.0010	0.0005	-0.0137	0.9875
12	-0.5319	-0.0032	-0.0011	-0.0013	-0.0012	-0.0033	-0.0149	0.9863
13	-0.5810	-0.0033	-0.0012	-0.0013	-0.0013	-0.0071	-0.0162	0.9851
14	-0.6301	-0.0035	-0.0012	-0.0013	-0.0013	-0.0109	-0.0175	0.9838
15	-0.6791	-0.0037	-0.0012	-0.0013	-0.0013	-0.0146	-0.0187	0.9825
16	-0.7281	-0.0039	-0.0012	-0.0013	-0.0012	-0.0184	-0.0200	0.9813
17	-0.7770	-0.0041	-0.0012	-0.0013	-0.0012	-0.0222	-0.0213	0.9800
18	-0.8259	-0.0043	-0.0012	-0.0013	-0.0012	-0.0259	-0.0225	0.9788
19	-0.8746	-0.0046	-0.0012	-0.0013	-0.0012	-0.0297	-0.0238	0.9775
20	-0.9234	-0.0048	-0.0012	-0.0013	-0.0012	-0.0335	-0.0250	0.9763

Quadro 4.29. - Resposta temporal calculada com a classe *statvar* (continuação).

Mais uma vez, para efeito de comparação e respeitando a precisão do números transcritos no quadro 4.29., foram obtidos resultados, através do MATLAB®, que aferiram exatamente com os descritos no quadro acima.

4.12.6. - Exemplo de Posicionamento de Pólos em Sistema de Equações de Estado

Dado o sistema de equações de estado linear e invariante no tempo, descrito pelas matrizes:

$$A=\begin{bmatrix}1 & 2 & 3\\ 3 & 2 & 1\\ 1 & 1 & 1\end{bmatrix},\qquad B=\begin{bmatrix}1 & 4\\ 2 & 5\\ 3 & 6\end{bmatrix},\qquad C=\begin{bmatrix}3 & 2 & 1\end{bmatrix}\qquad \text{e}\qquad D=\begin{bmatrix}0 & 0\end{bmatrix}$$

E constatando ainda, através dos recursos da própria biblioteca implementada, que os seus autovalores, nesse caso os pólos, encontram-se dados por 5, -1 e 0, decidiu-se então

realizar o posicionamento de pólos, sem descartar contudo o fato previamente conhecido do par (A, B) do sistema, acima descrito, ser completamente controlável; fato este que serve para viabilizar a aplicação efetiva do posicionamento de pólos [Wonham 79, Kautsky 85]. Os valores escolhidos para a alocação foram -0.2, -0.5 + 0.2j e -0.5 - 0.2j, com $j = \sqrt{-1}$.

Em vista destes fatos e com o uso da classe *statvar*, obteve-se uma matriz de realimentação estática de estados, a qual é dada por:

$$F = \begin{bmatrix} 1.65348 & 1.87027 & 1.41585 \\ -0.931378 & -1.04135 & -0.984881 \end{bmatrix}$$

Os recursos implementados permitiram também que a malha fosse fechada no sistema, utilizando-se da matriz F calculada e obtendo, portanto, uma nova matriz A', descrita por:

$$A' = A + BF = \begin{bmatrix} -1.07203 & -0.295134 & 0.476321 \\ 1.65007 & 0.533782 & -1.09272 \\ 0.372174 & 0.362698 & -0.661751 \end{bmatrix}$$

Por fim, uma reavaliação dos autovalores do sistema, mostrou os números desejados, isto é, a alocação procedeu-se devidamente nas posições -0.2, -0.5 + 0.2j e -0.5 - 0.2j.

4.13. - Conclusões

Os resultados obtidos denotam, na prática, que a meta do projeto foi alcançada. Tal fato não representa a finalização de uma biblioteca, mas ao contrário, apenas o início desta, uma vez que sempre existirá, a princípio, recursos passíveis de serem acrescentados. Inclusive, pesa significativamente sobre este aspecto, a convicção de que uma manutenção do investimento nesta linha de trabalho, trará benefícios em termos da produção de *software* no contexto de um simulador completo que, espera-se, venha a ser desenvolvido.

Neste capítulo foram expostas as nove classes de objetos implementadas que, em conjunto com os padrões de desenvolvimento e documentação realizados e já discutidos no capítulo anterior, respondem pela densidade de todo o projeto. Densidade esta que graduou desde

tipos considerados simples de realizar, como os vetores, até tipos mais elaborados como as funções de transferência e os sistemas de equações de estado.

Um número significativo de funções (e operadores) foi sucintamente descrito nos quadros de 4.1. até 4.25. A documentação detalhada de uso pode ser encontrada no LCMI ("Manuais de Utilização de Classe C++" de CL-0002 a CL-0010) [Suguieda 93b até Suguieda 93j].

Outrossim, foram também mostrados alguns exemplos para ajudar a situar o trabalho realizado. E por serem exemplos restritos, estes não abrangem todos os recursos descritos nas seções de 4.2. a 4.11., mas pelo menos ilustram uma parcial do potencial do produto implementado. Uma documentação mais abrangente pode ser encontrada nos já citados "Manuais de Utilização de Classe C++" de CL-0002 a CL-0010, onde para cada função (ou operador), um ou mais exemplos explicativos, visando justamente elucidar o respectivo uso e aplicação são apresentados.

4.14. - Referências Específicas

[Kautsky 85] KAUTSKY, J., NICHOLS, N. K., VAN DOOREN, P. Robust Pole Assignment in Linear State Feedback. *Int. J. Control*, v. 41, n. 5, p. 1129 - 1155, 1985.

[Suguieda 93b] SUGUIEDA, Márcio Heidi. Manual de Utilização de Classe C++: Classe dvector. *Manual Interno LCMI - CL-0002 - Revisão A*, p. 1 - 10, LCMI - UFSC, Florianópolis - S.C., jul. 1993.

[Suguieda 93c] SUGUIEDA, Márcio Heidi. Manual de Utilização de Classe C++: Classe dmatrix. *Manual Interno LCMI - CL-0003 - Revisão A*, p. 1 - 17, LCMI - UFSC, Florianópolis - S.C., jul. 1993.

[Suguieda 93d] SUGUIEDA, Márcio Heidi. Manual de Utilização de Classe C++: Classe extdmat. *Manual Interno LCMI - CL-0004 - Revisão A*, p. 1 - 11, LCMI - UFSC, Florianópolis - S.C., jul. 1993.

[Suguieda 93e] SUGUIEDA, Márcio Heidi. Manual de Utilização de Classe C++: Classe cvector. *Manual Interno LCMI - CL-0005 - Revisão A*, p. 1 - 11, LCMI - UFSC, Florianópolis - S.C., jul. 1993.

[Suguieda 93f] SUGUIEDA, Márcio Heidi. Manual de Utilização de Classe C++: Classe cmatrix. *Manual Interno LCMI - CL-0006 - Revisão A*, p. 1 - 18, LCMI - UFSC, Florianópolis - S.C., jul. 1993.

[Suguieda 93g] SUGUIEDA, Márcio Heidi. Manual de Utilização de Classe C++: Classe extcmat. *Manual Interno LCMI - CL-0007 - Revisão A*, p. 1 - 10, LCMI - UFSC, Florianópolis - S.C., jul. 1993.

[Suguieda 93h] SUGUIEDA, Márcio Heidi. Manual de Utilização de Classe C++: Classe polynom. *Manual Interno LCMI - CL-0008 - Revisão A*, p. 1 - 12, LCMI - UFSC, Florianópolis - S.C., jul. 1993.

[Suguieda 93i] SUGUIEDA, Márcio Heidi. Manual de Utilização de Classe C++: Classe trfunct. *Manual Interno LCMI - CL-0009 - Revisão A*, p. 1 - 24, LCMI - UFSC, Florianópolis - S.C., jul. 1993.

[Suguieda 93j] SUGUIEDA, Márcio Heidi. Manual de Utilização de Classe C++: Classe statvar. *Manual Interno LCMI - CL-0010 - Revisão A*, p. 1 - 22, LCMI - UFSC, Florianópolis - S.C., jul. 1993.

[Sun 89a] SUN MICROSYSTEMS. Sun® C++ Programmer's Guide. Mountain View - C.A., USA: Sun Microsystems, Inc., 1989.

[Sun 89d] SUN MICROSYSTEMS, AT&T. AT&T C++ Translator - Library Manual. USA: AT&T, 1989.

[Wonham 79] WONHAM, W. Murray. Linear Multivariable Control: a Geometric Approach. 2. ed. New York: Springer-Verlag New York Inc., 1979.

CAPÍTULO 5

CONCLUSÕES E PERSPECTIVAS

Concluir este trabalho significa dar início a mais uma nova etapa de uma longa jornada de desenvolvimento de ferramentas de auxílio ao engenheiro, em prática no Laboratório de Controle e Microinformática da UFSC. Aliás, uma das principais intenções deste projeto foi a de estipular e implementar uma metodologia de trabalho, por enquanto específica para a linguagem C++, que possibilite o aproveitamento de conceitos contemporâneos de produção de *software* e os destine à área de controle de processos. Esta idéia, a propósito, pode ser traduzida, simplesmente, na intenção concretizada de integração entre a Engenharia de Controle e a Engenharia de *Software*.

Alguns conceitos estudados pela Engenharia de *Software*, em particular os relativos a reutilização e a manutenção, tiveram um enfoque maior, devido a possibilidade e a potencialidade destes em contribuirem para otimizar tanto a produtividade como a qualidade de programas computacionais. Outrossim, em função também de uma igual possibilidade de minimização de custos, porém em termos mais provavelmente de médio e longo prazo, já que o desenvolvimento de módulos reutilizáveis requer naturalmente maiores recursos. O esforço de programação, visando reutilização, exige desde uma documentação apropriada até um controle de qualidade e de funcionalidade mais aprimorados.

Assim sendo, a meta fundamental do trabalho acaba se refletindo, então, na tentativa de minimizar, ao longo do tempo, o esforço de programação pela reutilização de módulos de programas desenvolvidos com esta finalidade. Módulos reutilizáveis que, por sua vez, passam a ser considerados como produtos finais e, por conseguinte, de uso mais abrangente do que aquele destinado a um pacote ou ambiente computacional único e específico. Isto inclusive faz com que uma maior atenção seja destinada à produção dos mesmos e, conseqüentemente, haja uma maior chance de reutilização, assim como de manutenção. De fato, o fundamento desta idéia vem justamente ao encontro da famosa questão: "Para que reinventar a roda?".

Esta ênfase em abordar os temas produto e produção vem apropriadamente da crescente necessidade de integração do meio universitário com o industrial [Bristol 86], porquanto a realidade atual mostra que, à medida que transcorre a evolução do ambiente

acadêmico, principalmente em relação à área tecnológica, os vocábulos produtividade, qualidade e custo se firmam, cada vez mais, no cotidiano dos profissionais envolvidos nesse meio.

Ainda sob o aspecto de reutilização, aparece outro ponto de interesse neste projeto, cuja caracterização pode ser compreendida no incentivo ao espírito de cooperatividade, face ao de competitividade, relativos ao ambiente acadêmico. A valorização deste ponto é importante, porque é bastante freqüente a perda de rotinas implementadas no meio acadêmico brasileiro, o que acarreta na necessidade de repetição de esforço de programação e de manutenção.

Para que esta linha de trabalho possa ter uma continuidade, todo um conjunto de especificações foi detalhadamente esmerado ao longo deste projeto. Obviamente, por ser recente, ainda não há como garantir a real eficácia das especificações adotadas, mas pelo menos foram apresentadas as justificativas para a definição destas e, se necessário, uma avaliação prévia pode ser feita por outros usuários que trabalhem em linhas de pesquisa e desenvolvimento semelhantes.

Estas especificações, de fato, abordaram desde conceitos da Engenharia de *Software* até detalhes de programação na linguagem C++. Em função primordialmente dos propósitos de trabalho e por motivos também de tempo, todas estas foram ensaiadas apenas para o compilador Sun® C++ 2.1, instalado em estações de trabalho Sun® SPARC, as quais, por sua vez, operavam sob o sistema operacional UNIX®. Inclusive, convém salientar que a portabilidade não mereceu atenção especial neste etapa de trabalho, porque a decisão inicial de desenvolver todo o projeto em estações de trabalho já predisse a ausência do fator portabilidade como prioridade.

Por outro lado, quanto a qualidade do material já implementado, houve todo um trabalho de verificação e validação de cada rotina implementada. Mesmo assim, por ser demasiado cedo, não há como avaliar nesse momento, a real qualidade do mesmo, pois somente o uso poderá fornecer tal informação.

Além disso, cabe salientar que a suscetibilidade a erros existe em todo protótipo de *software* e, portanto, este projeto não está isento a tal fator. Esta suscetibilidade, neste caso, é concreta, primordialmente devido ao número de linhas de código envolvidas e também porque este projeto teve as etapas de implementação e teste realizadas por um único programador. O quadro 5.1., na página subseqüente, apresenta uma estatística do número de linhas envolvidas apenas com a codificação, isto é, este quadro não abrange as linhas de código reutilizadas automaticamente dos pacotes LINPACK e EISPACK.

	arquivo de *inteface* (*.h)	arquivo de implemen-tação (*.cc)	arquivo de teste (tst*.cc)	arquivo de compilação e linkagem (makefile)	SUBTOTAL
classe *dvector*	69	314	261	57	701
classe *dmatrix*	104	887	417	57	1465
classe *extdmat*	75	314	455	114	958
classe *cvector*	75	358	286	57	776
classe *cmatrix*	107	804	434	83	1428
classe *extcmat*	67	249	493	111	920
classe *polynom*	78	590	280	57	1005
classe *trfunct*	142	938	502	57	1639
classe *statvar*	130	1185	643	89	2047
SUBTOTAL	847	5639	3771	682	10939

Quadro 5.1. - Parciais do número de linhas codificadas no projeto[1].

A ausência, comum à realidade acadêmica, de indivíduos ou grupos, especialmente voltados ao controle de qualidade ou à homologação de *software*, aumenta a probabilidade do aparecimento de erros. Isto inclusive colabora para uma predisposição no sentido de evitar o uso de protótipos ou programas em desenvolvimento, principalmente os do meio acadêmico. Esses aspectos e o esforço de manutenção de um *software*, estimado atualmente em cerca de 70% do esforço total de desenvolvimento do produto [Fairley 85, Griss 91], ajudam a justificar a valorização do tópico manutenção neste trabalho.

Do ponto de vista da área de controle, foram implementadas nove classes de objetos cujo escopo cobre os tipos de dados abstratos selecionados para iniciar a biblioteca especificada. Estes tipos - vetores, matrizes, polinômios, funções de transferência e sistemas de equações de estado - já abrangem uma representação apropriada dos dados, assim como métodos de manipulação e análise dos mesmos. Isto tudo ainda, esmerado sob uma estrutura considerada simples de utilizar e de realizar, mostrando que este projeto merece ter uma continuidade, ou seja, já há disponível uma metodologia para que outros recursos possam ser adicionados, permitindo assim que o nível da produção de *softwares* no LCMI seja otimizado.

De acordo com o implementado, foi ressaltado também a utilidade das classes de objetos criadas, tanto para os programas de uso pessoal e específico como para o desenvolvimento de ferramentas de auxílio ao engenheiro de controle. Neste último caso,

[1] O quadro apresenta o número de linhas realmente programadas, isto é, não está incluso o número de linhas obtidas diretamente com a reutilização. Por outro lado, salienta-se que a contagem foi realizada com base no número de linhas dos arquivos e, por isso, foram também contabilizadas as linhas de comentários e as linhas em branco.

estabeleceu-se a relação da biblioteca elaborada com os chamados sistemas *CACE* (*Computer Aided Control Engineering*), situando assim, o projeto como base para o desenvolvimento de ferramentas de auxílio ao engenheiro - os denominados projetos PACSC (Projeto Assistido por Computador para a área de Sistemas de Controle).

No caso do desenvolvimento de um simulador para análise e controle de processos, fica a expectativa de trabalhos futuros que envolvam:

- o uso e a evolução da biblioteca básica implementada;
- o desenvolvimento de biblioteca específica que englobe outras particularidades da área de controle de processos;
- a estruturação das bases de métodos e de dados (sistemas *CACE*) para o simulador;
- a *interface* com o usuário provida de recursos gráficos e com, preferencialmente, uma linguagem orientada para a área de controle; e,
- o supervisor/interpretador responsável pela coordenação (ou gerenciamento) dos recursos do sistema.

A propósito, em paralelo a este trabalho, decorreu-se um outro projeto de mestrado que, basicamente, já procura abranger alguns dos aspectos acima citados.

Convém por fim lembrar que está associado diretamente ao trabalho apresentado, todo um conjunto de manuais criteriosamente planejados e desenvolvidos, no intuito de promover o uso devido e correto das classes de objetos implementadas. Classes estas que moldam "**Uma Biblioteca de Tipos Abstratos de Dados para a Área de Análise e Projeto de Sistemas de Controle**".

5.1. - Referências Específicas

[Bristol 86] BRISTOL, Edgar H. An Industrial Point of View on Control Teaching and Theory. *IEEE Control Systems Magazine*, v. 6, p. 24 - 27, fev. 1986.

[Fairley 85] FAIRLEY, Richard E. Software Engineering Concepts. Singapore: McGraw-Hill Book Company, 1985.

[Griss 91] GRISS, Martin L., ADAMS, Sam S., BAETJER Jr., Howard, COX, Brad J., GOLDBERG, Adele. The Economics of Software Reuse. *Proceedings of OOPSLA'91 - Object-Oriented Programming Systems, Languages, and Applications*, Phoenix, Arizona, v. 26, n. 11, p. 264 - 270, maio 1993.

REFERÊNCIAS BIBLIOGRÁFICAS

[Aström 83] ÅSTRÖM, Karl Johan. Computer Aided Modeling, Analysis and Design of Control Systems - A Perspective. *IEEE Control System Magazine*, v. 3, p. 4 - 16, maio 1983.

[Aström 85] ÅSTRÖM, Karl Johan. Process Control - Past, Present and Future. *IEEE Control Systems Magazine*, v. 5, p. 3 - 10, ago. 1985.

[Bristol 86] BRISTOL, Edgar H. An Industrial Point of View on Control Teaching and Theory. *IEEE Control Systems Magazine*, v. 6, p. 24 - 27, fev. 1986.

[Bueno 91] BUENO, Francisco da Silveira. Minidicionário da Língua Portuguesa. 5. ed. atualizada. São Paulo - S.P.: Editora Lisa Ltda, 1991.

[De Pieri 93] DE PIERI, E. R., ABOU-KANDIL, H., SUGUIEDA, M. H. Sensor Positioning for Large Flexible Structures. *Proceedings of Structural Optimization 93: The World Congress on Optimal Design of Structural Systems*, Rio de Janeiro, v. , n. , p. , ago. 1993.

[Deitel 84] DEITEL, Harvey M. An Introduction to Operating Systems. 1. ed. revisada. Massachusetts, USA: Addison-Wesley Publishing Company, 1984.

[Diehl 86] DIEHL, Kent S., KROGH, Bruce H., NAGURKA, Mark L. An Interactive Control Systems Simulator. *IEEE Control Systems Magazine*, v. 6, p. 20 - 26, abr. 1986.

[Dongarra 79] DONGARRA, J. J., MOLER, C. B., BUNCH, J. R., STEWART, G. W. LINPACK: User's Guide. Philadelphia, USA: SIAM - Society for Industrial and Applied Mathematics, 1979.

[Eckel 91] ECKEL, Bruce. C++ Guia do Usuário. Tradução de Luis Antonio Fontes Quintela, revisão técnica de Edison Raymundi Júnior. São Paulo - S.P.: Makron Books do Brasil Editora Ltda. e Editora McGraw-Hill, 1991.

[Fairley 85] FAIRLEY, Richard E. Software Engineering Concepts. Singapore: McGraw-Hill Book Company, 1985.

[Farines 86] FARINES, Jean-Marie, SAVI, Vanio M., BRUCIAPAGLIA, Augusto H. Projeto Assistido por Computador para Sistemas de Controle: Um Pacote Interativo. *Anais do 6º. CBA - Congresso Brasileiro de Automática*, UFMG, Belo Horizonte - M.G., v. 1, p. 550 - 554, nov. 1986.

[Ferreira 87] FERREIRA, P. A. V., FONTANINI, W., GUERRA, A. C., AMARAL, W. C., GOMIDE, F. A. C., Modelagem, Análise e Projeto de Sistemas de Dinâmicos Integrados por Computador. *Revista SBA: Controle & Automação*, v. 1, n. 4, p. 322 - 330, out. 1987.

[Fontanini 90] FONTANINI, W., SILVA Fº., O. S., FERREIRA, P. A. V. Um Ambiente Integrado para Análise e Projeto no Espaço de Estados. *Anais do 8º. CBA - Congresso Brasileiro de Automática*, UFPa, Belém - P.A., v. 1, p. 189 - 194, 1990.

[Griss 91] GRISS, Martin L., ADAMS, Sam S., BAETJER Jr., Howard, COX, Brad J., GOLDBERG, Adele. The Economics of Software Reuse. *Proceedings of OOPSLA'91 - Object-Oriented Programming Systems, Languages, and Applications*, Phoenix, Arizona, v. 26, n. 11, p. 264 - 270, maio 1993.

[Holliker 90] HOLLIKER, William. UNIX® Shell Commands Quick Reference. Carmel, Indiana, USA: Que® Corporation, 1990.

[Jamshidi 85] JAMSHIDI, M., HERGET, C. J. (editores). Computer Aided Control Systems Engineering. New York - N.Y., USA: North-Holland/Elsevier Science Publishing Company, 1985.

[Kaplan 92] KAPLAN, Gadi. Talking About Tools. *IEEE Spectrum*, v. 29, n. 11, p. 32 - 33, nov. 1992.

[Kautsky 85] KAUTSKY, J., NICHOLS, N. K., VAN DOOREN, P. Robust Pole Assignment in Linear State Feedback. *Int. J. Control*, v. 41, n. 5, p. 1129 - 1155, 1985.

[Kelly-Bootle 89] KELLY-BOOTLE, Stan. Dominando o Turbo C. Tradução de Eduardo Alberto Barbosa. 2. ed. Rio de Janeiro - R.J.: Editora Ciência Moderna, 1989.

[Lewis 91] LEWIS, John A., HENRY, Sallie M., KAFURA, Dennis G., SCHULMAN, Robert S. An Empirical Study of the Object-Oriented Paradigm and Software Reuse. *Proceedings of OOPSLA'91 - Object-Oriented Programming Systems, Languages, and Applications*, Phoenix, Arizona, v. 26, n. 11, p. 184 - 196, nov. 1991.

[Lipaev 82] LIPAEV, V. V. Computer-Aided Design of Software for Control Systems. *Computers in Control*, Plenum Publishing Corporation, p. 212 - 221, 1982.

[MathWorks 91] MATHWORKS. MATLAB™ High-Performance Numeric Computation Software for 398/486 MS-DOS Computers: 386-MATLAB User's Guide. The MathWorks, Inc., 1991.

[Meyer 88] MEYER, Bertrand. Object-oriented Software Construction. Cambridge, Great Britain: Prentice Hall, 1988.

[Michaelis 89] MICHAELIS: Pequeno Dicionário Inglês-Português Português-Inglês. Ed. revista e atualizada. São Paulo - S.P.: Comp. Melhoramentos de São Paulo, Indústrias de Papel, 1989.

[Montanaro 90] MONTANARO, George D., FREDERICK, Dean K. Workstations as Environments for the Analysis and Design of Controls Systems. *IEEE Control Systems Magazine*, v. 10, n. 3, p. 114 - 121, abr. 1990.

[Naitoh 85] NAITOH, Haruo. Control System Design CAD System on a Personal Computer. *IECON'85 - Conference of IEEE Industrial Electronics Society*, p. 41 - 46, 1985.

[Pappas 91] PAPPAS, Chris H., MURRAY III, William H. Turbo C++ Completo e Total. Tradução de Mário Moro Fecchio, revisão técnica de José Eduardo Maluf de Carvalho. São Paulo - S.P.: Makron Books do Brasil Editora Ltda e Editora McGraw-Hill Ltda., 1991.

[Press 87] PRESS, William H. FLANNERY, Brian P. TEUKOLSKY, Saul A., VETTERLING, William T. Numerical Recipes: The Art of Scientific Computing. New York - N.Y., USA: Cambridge University Press, 1987.

[Pressman 87] PRESSMAN, Roger S. Software Engineering: A Practitioner's Approach. 2. ed. Singapore: McGraw-Hill Book Company, 1987.

[Ralston 76] RALSTON, Anthony (editor). Encyclopedia of Computer Science. 1. ed. New York - N.Y., USA: Van Nostrand Reinhold Company, 1976.

[Ranéa 90] RANÉA, Pierre-Guy, MARQUES, Fernanda Isabel. Manual dos Comandos Usuais DOS/UNIX. *Nota Técnica LCMI 90/4*, LCMI - UFSC, Florianópolis - S.C., maio 1990.

[Savi 88] SAVI, Vânio M., CASTELAN N., Eugênio B., BRUCIAPAGLIA, Augusto H., FARINES, Jean-Marie. Uma Visão sobre o Desenvolvimento de Pacotes de Projeto Assistido por Computador para Sistemas de Controle. *Anais do 7º. CBA - Congresso Brasileiro de Automática*, ITA, São José dos Campos - S.P., v. 1, p. 173 - 178, 1988.

[Schildt 91] SCHILDT, Herbert. C Completo e Total. Tradução de Marcos Ricardo Alcântara de Moraes. São Paulo - S.P.: Makron Books do Brasil Editora Ltda., 1991.

[Silva 90] SILVA, Maurício A., GOZZI, Jomar. LABCON - Software para Análise, Projeto e Simulação de Sistemas de Controle. *Anais do 8º. CBA - Congresso Brasileiro de Automática*, UFPa, Belém - PA, v. 1, p. 208 - 213, 1990.

[Singnoretti 90] SINGNORETTI, A., GOMIDE, F., BINGULAC, S. Um Ambiente Baseado em Conhecimento para Projeto de Sistemas de Controle. *Anais do 8º. CBA - Congresso Brasileiro de Automática*, UFPa, Belém - PA, v. 1, p. 182 - 188, 1990.

[Spang III 84] SPANG III, H. Austin. The Federated Computer-Aided Control Design System. *Proceedings of the IEEE*, v. 72, n. 12, p. 1724 - 1731, dez. 1984.

[Spang III 85] SPANG III, H. Austin. Experience and Future Needs in Computer-Aided Control Design. *IEEE Control Systems Magazine*, v. 5, p. 18 - 21, fev. 1985.

[Suguieda 90] SUGUIEDA, Márcio Heidi. Experiências de Laboratório de Controle Digital. (Projeto Final de Graduação em Engenharia Elétrica.) Brasília - D.F.: Universidade de Brasília - UnB, 1990.

[Suguieda 93a] SUGUIEDA, Márcio Heidi, KAMMER, Leonardo César. Norma para Documentação de Classes C++. *Publicação Interna LCMI PI 93-1*, LCMI - UFSC, Florianópolis - S.C., jun. 1993.

[Suguieda 93b] SUGUIEDA, Márcio Heidi. Manual de Utilização de Classe C++: Classe dvector. *Manual Interno LCMI - CL-0002 - Revisão A*, p. 1 - 10, LCMI - UFSC, Florianópolis - S.C., jul. 1993.

[Suguieda 93c] SUGUIEDA, Márcio Heidi. Manual de Utilização de Classe C++: Classe dmatrix. *Manual Interno LCMI - CL-0003 - Revisão A*, p. 1 - 17, LCMI - UFSC, Florianópolis - S.C., jul. 1993.

[Suguieda 93d] SUGUIEDA, Márcio Heidi. Manual de Utilização de Classe C++: Classe extdmat. *Manual Interno LCMI - CL-0004 - Revisão A*, p. 1 - 11, LCMI - UFSC, Florianópolis - S.C., jul. 1993.

[Suguieda 93e] SUGUIEDA, Márcio Heidi. Manual de Utilização de Classe C++: Classe cvector. *Manual Interno LCMI - CL-0005 - Revisão A*, p. 1 - 11, LCMI - UFSC, Florianópolis - S.C., jul. 1993.

[Suguieda 93f] SUGUIEDA, Márcio Heidi. Manual de Utilização de Classe C++: Classe cmatrix. *Manual Interno LCMI - CL-0006 - Revisão A*, p. 1 - 18, LCMI - UFSC, Florianópolis - S.C., jul. 1993.

[Suguieda 93g] SUGUIEDA, Márcio Heidi. Manual de Utilização de Classe C++: Classe extcmat. *Manual Interno LCMI - CL-0007 - Revisão A*, p. 1 - 10, LCMI - UFSC, Florianópolis - S.C., jul. 1993.

[Suguieda 93h] SUGUIEDA, Márcio Heidi. Manual de Utilização de Classe C++: Classe polynom. *Manual Interno LCMI - CL-0008 - Revisão A*, p. 1 - 12, LCMI - UFSC, Florianópolis - S.C., jul. 1993.

[Suguieda 93i] SUGUIEDA, Márcio Heidi. Manual de Utilização de Classe C++: Classe trfunct. *Manual Interno LCMI - CL-0009 - Revisão A*, p. 1 - 24, LCMI - UFSC, Florianópolis - S.C., jul. 1993.

[Suguieda 93j] SUGUIEDA, Márcio Heidi. Manual de Utilização de Classe C++: Classe statvar. *Manual Interno LCMI - CL-0010 - Revisão A*, p. 1 - 22, LCMI - UFSC, Florianópolis - S.C., jul. 1993.

[Sun 89a] SUN MICROSYSTEMS. Sun® C++ Programmer's Guide. Mountain View - C.A., USA: Sun Microsystems, Inc., 1989.

[Sun 89b] SUN MICROSYSTEMS, AT&T. AT&T C++ Language System - Reference Manual. USA: AT&T, 1989.

[Sun 89c] SUN MICROSYSTEMS, AT&T. AT&T C++ Language System - Selected Readings. USA: AT&T, 1989.

[Sun 89d] SUN MICROSYSTEMS, AT&T. AT&T C++ Translator - Library Manual. USA: AT&T, 1989.

[Sun 90a] SUN MICROSYSTEMS. Sun® FORTRAN Reference Manual. Mountain View - C.A. , USA: Sun Microsystems, Inc., 1990.

[Sun 90b] SUN MICROSYSTEMS. Sun® FORTRAN User's Guide. Mountain View - C.A., USA: Sun Microsystems, Inc., 1990.

[Sutherland 85] SUTHERLAND, Hunt A., SONIN, Karen L. Control Engineers Workbench - A Methodology for Microcomputer Implementation of Controls. *IEEE Control Systems Magazine*, v. 5, p. 22 - 26, fev. 1985.

[Takahashi 88] TAKAHASHI, Tadao. Introdução a Programação Orientada a Objetos. Edição EBAI. Curitiba - P.R.: III EBAI, jan. 1988.

[Tozzi 86] TOZZI, Clésio Luis. PAC: Projeto Auxiliado por Computador. Campinas - S.P.: Editora da Unicamp e Editora Papirus, 1986.

[Wonham 79] WONHAM, W. Murray. Linear Multivariable Control: a Geometric Approach. 2. ed. New York: Springer-Verlag New York Inc., 1979.

ANEXO I

NORMA SUGERIDA PARA DOCUMENTAÇÃO (E DESENVOLVIMENTO PADRÃO) DE CLASSES C++

LCMI

LABORATÓRIO DE CONTROLE E MICROINFORMÁTICA

"Prof. Marcos Cardoso Filho"

NORMA PARA DOCUMENTAÇÃO DE CLASSES C++

Márcio Heidi Suguieda

Leonardo César Kammer

Publicação Interna PI 93-1

Laboratório de Controle e Microinformática "Prof. Marcos Cardoso Filho"

Universidade Federal de Santa Catarina, Departamento de Engenharia Elétrica

Campus Universitário - 88040-900 Florianópolis SC BRASIL

Tel: +55 (482) 31-9202 - Fax: +55 (482) 341524 - E-mail: lcmi@brufsc.bitnet

I.1. - Introdução

Esta publicação interna tem por objetivo normatizar a documentação de classes desenvolvidas na linguagem de programação orientada a objetos C++. Para tanto são descritos, a seguir, os procedimentos recomendados para que se possa uniformizar tanto a implementação quanto a documentação destas classes.

Cada classe devidamente documentada passa a ser referenciada por um código do tipo **CL-*XXXX***, onde ***XXXX*** corresponde a um número de quatro dígitos, como por exemplo **CL-0026**. Ao final deste documento encontra-se uma lista das classes implementadas, na qual devem ser adicionadas as informações básicas de cada nova implementação.

Inicialmente, na seção I.2., são discutidas as padronizações em termos dos arquivos de implementação. Na seção I.3. apresenta-se as padronizações relativas à documentação da classe, nas formas: Manual de Utilização e "man pages" (específico para UNIX). Cumpridas as etapas descritas nas seções I.2. e I.3., procede-se à organização da nova classe (seção I.4.) de modo a garantir uma utilização comum.

I.2. - Implementação

A fim de padronizar os arquivos de implementação da classe, recomenda-se a utilização de formatos pré-estabelecidos, tanto para o arquivo "header" (.h) quanto para o arquivo contendo o corpo da implementação (.cc). Estes arquivos pré-estabelecidos estão voltados ao desenvolvimento em ambiente UNIX, com a utilização das ferramentas "make" e "sccs". O uso destas ferramentas é fortemente recomendada como apoio ao desenvolvimento.

Nas seções I.2.1., I.2.2. e I.2.3. são apresentados os arquivos "padrao.h", "padrao.cc" e "makefile", respectivamente, com suas descrições correspondentes. Estes arquivos podem ser encontrados no subdiretório **Class++/src** da área de produtos existente na rede UNIX do LCMI. Na seção I.2.4. encontram-se algumas sugestões a serem seguidas durante o desenvolvimento da classe.

I.2.1. - Formato Padrão do Arquivo *Header*

A listagem do arquivo "padrao.h" pode ser vista a seguir. Recomenda-se fazer uma cópia para sua área de trabalho e iniciar a implementação da classe a partir do modelo dado, substituindo-se os campos concernentes.

Todos os códigos da forma %*X*%, onde *X* corresponde a uma letra maiúscula, são relativos à ferramenta "sccs", devendo seu manual ser consultado para um melhor entendimento: "Programmer's Overview Utilities & Libraries", seção "Programming Utilities & Libraries", capítulo 4, SUN Microsystems, 1990.

Todos os campos delimitados por "<" e ">" devem ser substituídos por dados pertinentes. O caracter X indica um número a ser devidamente preenchido.

No campo AUTOR(ES) deve-se informar o nome do programador responsável pela criação da classe, seguido opcionalmente pelos nomes dos programadores que venham a alterar características desta. As informações V*X*.*X* e Mes/Ano indicam a versão a partir da qual as modificações do referido autor iniciaram-se.

```
// *** %W% (%G%) LCMI
// ***************************************************************************
//
//              (c) Copyright 199X - <Nome do autor> - LCMI/EEL/UFSC
//
//              AUTOR(ES): <Nome do autor (criador) - Mes/Ano>
//                         <Nome do autor de modificacoes - VX.X - Mes/Ano>
//
//              HISTORICO: %G% (mm/dd/aa) - versao %I%
//
//              LINGUAGEM: Sun C++ 2.1
//
//              PROPOSITO: <descricao resumida>
//
//              OBSERVACOES:
//
// ***************************************************************************

#ifndef <identificador>
#define <identificador>

class padrao {
        padrao();
        // ...
        virtual ~padrao();
        // ...

  protected:
        // ...
```

Figura I.1. - Formato padrão do arquivo *header* ou arquivo de *interface*.

```
  private:
        // ...
};

#endif // <identificador>
```

Figura I.1. - Formato padrão do arquivo *header* ou arquivo de *interface* (continuação).

I.2.2. - Formato Padrão do Arquivo de Corpo

A listagem do arquivo "padrao.cc" pode ser vista a seguir. Recomenda-se fazer uma cópia para sua área de trabalho e iniciar a implementação da classe a partir do modelo dado, substituindo-se os campos concernentes.

```
// ***************************************************************************
//
//              (c) Copyright 199X - <Nome do autor> - LCMI/EEL/UFSC
//
//              AUTOR(ES): <Nome do autor (criador) - Mes/Ano>
//                         <Nome do autor de modificacoes - VX.X - Mes/Ano>
//
//              HISTORICO: %G% (mm/dd/aa) - versao %I%
//
//              OBSERVACOES:
//
// ***************************************************************************

#include <padrao.h>

????
padrao::????(????) {

#ifndef __MSDOS__
 // Identificador para o comando "what" (UNIX):
        static char sccsid[] = "%W% (%G%) LCMI";
#endif // __MSDOS__

        <corpo da funcao membro>

}
```

Figura I.2. - Formato padrão do arquivo de corpo ou arquivo de implementação.

De forma semelhante ao arquivo "padrao.h", os códigos %*X*% são relativos à ferramenta "sccs". Igualmente todos os campos delimitados por "<" e ">", e os "X" devem ser substituídos. As seqüências "????" são utilizadas somente para indicar a existência de seqüências de caracteres adequados.

Salienta-se que no corpo de uma das funções, preferencialmente a primeira, define-se o identificador *sccs* para o comando "what" do sistema UNIX.

I.2.3. - Formato Padrão do Arquivo *Makefile*

A listagem do arquivo "makefile" pode ser vista a seguir. Recomenda-se fazer uma cópia para sua área de trabalho e alterá-la em função da implementação da classe.

```
OBJECTS = <padrao.o ...>

INCLUDES = <path/include/padrao.h ...>

LIBRARY = <path/lib/libpadrao.a>

EXEC = <testpadrao>

CCFLAGS += -I<path>/include
LDFLAGS += -L<path>/lib

#.SILENT:
.KEEP_STATE:

.INIT: <padrao.h>

.PRECIOUS: $(LIBRARY)

all: $(LIBRARY) $(INCLUDES)

$(LIBRARY): $(OBJECTS)
        ar rv $@ $?
        ranlib $@

<path>/include/%.h: %.h
        rm -f $@
        cp $< $@

debug: $(EXEC)

debug := CCFLAGS += -g -pg

$(EXEC): $(EXEC).o $(OBJECTS)
        $(LINK.cc) $@.o $(OBJECTS) -o $@ <-l????>

clean:
        rm -f *.o *% $(EXEC) *.out .nfs*
        sccs clean
```

Figura I.3. - Formato padrão do arquivo *makefile*.

A substituição de campos é similar a dos arquivos "padrao.h" e "padrao.cc". Para maiores informações a respeito da ferramenta "make", deve-se consultar o manual: "Programmer's Overview Utilities & Libraries", seção "Programming Utilities & Libraries", capítulo 5, SUN Microsystems, 1990.

I.2.4. - Recomendações Gerais

- Para facilitar a utilização de terminais e impressoras, enfatiza-se a importância de limitar o comprimento de cada linha em 80 colunas;

- A utilização deste padrão no sistema operacional DOS é viável, devendo os códigos de controle da ferramenta "sccs" serem substituídos ou removidos, conforme o caso;

- A utilização de diferentes compiladores C++ pode tornar necessária a alteração da extensão ".cc", sendo esta a extensão padrão recomendada para desenvolvimento no sistema UNIX;

- Recomenda-se a inclusão de comentários nos arquivos ".h" e ".cc". Estes comentários devem estar voltados ao funcionamento interno das rotinas. As informações concernentes à utilização da classe deverão estar contidas no Manual de Utilização descrito na seção I.3. desta publicação interna;

I.3. - Documentação

I.3.1. - Manual de Utilização

A documentação das classes deve seguir um padrão pré-definido mostrado no anexo I.a. Para os usuários do editor "Word for Windows", este padrão está disponível no LCMI na forma de um arquivo "template" denominado **CLASSCPP.DOT**. Todavia, qualquer editor de texto pode ser utilizado para a confecção deste documento, denominado "Manual de Utilização de Classe C++".

Cada uma das seções do Manual de Utilização é descrita a seguir.

• Denominação da Classe

Toda classe passará a conter um código do tipo CL-*XXXX*, conforme seqüência apresentada na seção I.4. desta publicação interna. Uma classe que seja herdeira de outra(s) deve informar o(s) nome(s) da(s) classe(s) base.

Deve-se informar a portabilidade entre os sistemas DOS e UNIX, onde o código "S" implica na portabilidade, o código "N" implica na não portabilidade e o código "?" indica a não verificação desta informação. Além da compatibilidade em termos de possibilidade de compilação, deve-se informar a possibilidade de transferência de arquivos de dados, gravados pela classe, de um sistema operacional para outro. Caso não haja leitura ou escrita de dados em arquivos, utiliza-se o código "-".

A genericidade de uma classe deve ser indicada. Uma classe genérica é aquela independente de recursos de software ou hardware específicos, tais como ambientes gráficos ou co-processadores. A legenda abaixo, passível de expansão, identifica os diversos níveis de genericidade das classes:

- (G) : Genérico
- (X) : XView/OpenWindows (UNIX)
- (W) : Windows (DOS)
- (O) : ObjectWindows (Borland C++, Windows)
- (C) : Co-processador numérico
- (__) : _____
- (__) : _____
- (__) : _____
- (__) : _____
- (__) : _____
- (__) : _____

Uma classe normalmente divide-se em dois arquivos, em termos de utilização - o arquivo "header" e o arquivo com o código objeto - devendo este último estar inserido em uma biblioteca. No campo "Parâmetros Mínimos de Linkagem" devem ser colocados os parâmetros necessários à linkagem do código da classe com as demais partes integrantes do programa.

Os itens "Autor(es)" e "Versão" referem-se à versão da classe documentada.

• Descrição Geral

Este item está auto-documentado no arquivo "template" (ver anexo I.a.).

• Membros da Classe

Basicamente as informações necessárias neste item estão auto-documentadas no anexo I.a. Convém salientar que as funções (e operadores) herdadas e não alteradas receberão apenas uma documentação simples, indicando o seu propósito e a localização das informações complementares (no caso o Manual de Utilização da classe base).

Recomenda-se agrupar as funções (e operadores) sobrecarregadas em uma única descrição, pois o propósito destas funções tende a ser o mesmo. Funções construtoras nem sempre obedecem a regra anterior, devendo ser consideradas caso a caso e agrupadas de acordo com o bom senso do programador.

I.3.2. - "*Man Pages*"

As "man pages" são manuais de consulta rápida, invocados pelo comando "man" do sistema operacional UNIX. Para a confecção destes manuais deve-se seguir uma codificação padrão para formatação de texto. Os códigos mais importantes estão presentes no exemplo mostrado nesta seção. Um conhecimento mais aprofundado destes e de outros códigos pode ser obtido através do manual: "SunOS Documentation Tools", seções "Using NROFF & TROFF" e "Formatting Documents", SUN Microsystems, 1990.

De forma correspondente aos arquivos "padrao.h" e "padrao.cc", encontra-se disponível no subdiretório **Class++/man/man3** da área de produtos o arquivo "padrao.3", o qual contém os campos básicos para criação de "man pages" para as classes C++ CL-XXXX. Este arquivo é apresentado a seguir, podendo o texto formatado ser visto através do comando *man padrao*.

```
.TH <PADRAO> 3C++ "DD MES 199X" "LCMI" "Classe C++ : CL-XXXX"
.SH NOME
<padrao \- proposito da classe (resumido)>
.SH SINOPSE
.nf
#include <padrao.h>

class padrao {
   public:
      ????
   protected:
      ????
};
.fi
.SH DESCRICAO
As funcoes desta classe podem ser encontradas
em "libpadrao.a", devendo esta biblioteca ser "linkada"
com os demais modulos atraves da opcao -lpadrao.
As declaracoes para esta classe
sao encontradas no arquivo de inclusao <padrao.h>.
Em funcao destas consideracoes, torna-se necessario
utilizar as opcoes -I<path>/include na compilacao
e -L<path>/lib na linkagem, onde <path> e'
o caminho para a area onde estao armazenadas
as classes CL-XXXX.
.PP
< Descricao geral da classe, apresentando
opcionalmente algumas (ou todas) funcoes e/ou
construtores, conforme sejam necessarios.
OBS: APAGAR ESTE PARAGRAFO NA VERSAO FINAL >
.SH CONSULTAR
< itens a consultar,
exemplo:
????(3C++),
????(3C++) >
```

Figura I.4. - Formato padrão do arquivo gerador das *man pages*.

Todas as linhas que iniciam com um ponto (.) seguido por dois caracteres correspondem a códigos para formatação do texto. Diversas partes deste arquivo devem ser substituídas, de forma análoga aos arquivos de implementação. Em caso de dúvida basta consultar "man pages" de classes já documentadas.

A extensão ".3" utilizada, bem como sua localização no subdiretório **man3**, respeita uma padronização estabelecida para as "man pages". De fato, a categoria 3 corresponde às funções de biblioteca ("SunOS Reference Manual" Vol. II).

I.4. - Organização das Classes C++

Após **concluídos** o desenvolvimento e a documentação de uma nova classe, deve-se torná-la disponível aos usuários e catalogá-la. Para permitir a utilização comum das classes C++, destina-

se uma cópia destas à área de produtos da rede UNIX, mais precisamente ao subdiretório **Class++**. A arborescência deste subdiretório é apresentada a seguir.

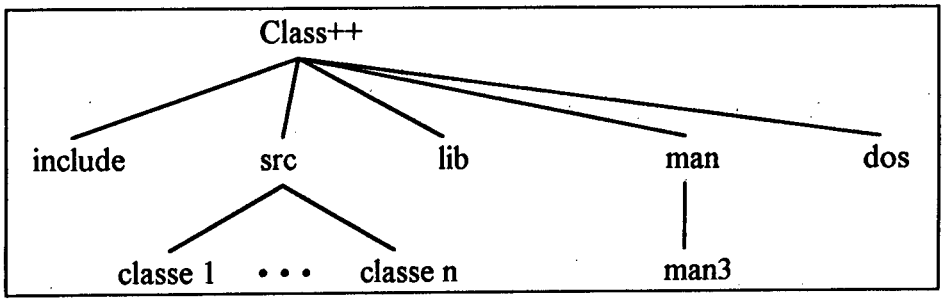


Figura I.5. - Arborescência dos subdiretórios quanto à organização das classes C++.

No subdiretório **include** encontram-se os arquivos header (.h). Em **src** encontram-se os subdiretórios com os arquivos (.h, .cc, teste, makefile, SCCS, etc.) de desenvolvimento da classe. O subdiretório **lib** contém as bibliotecas com os códigos objetos. Em **man/man3** são encontrados os arquivos de documentação ("man pages") das classes. Finalmente, as implementações realizadas no sistema DOS, e não testadas no sistema UNIX, estarão disponíveis no subdiretório **dos**, em uma arborescência semelhante a do subdiretório **src**.

O procedimento para catalogar uma classe consiste em preencher os campos correspondentes à lista existente no anexo I.b. Os campos a serem preenchidos equivalem aos do Manual de Utilização, exceto o campo PROPÓSITO que corresponde à descrição resumida contida no arquivo header.

ANEXO I.a. - FORMATO PADRÃO PARA DOCUMENTAÇÃO DE CLASSES C++

1. DENOMINAÇÃO DA CLASSE

- CL-XXXX: <Nome da Classe>
- Classe(s) Base: <Nome da(s) Classe(s), se houver(em), (public | private)>
- Portabilidade: DOS [?] UNIX [?] **(S/N/?)**
- Compatibilidade de Transferência de Arquivos entre DOS e UNIX: [?] **(S/N/?/-)**
- Genericidade: [?] **(G/X/W/O/C/...)**
- Header: <XXXX.h>
- Biblioteca: <xxxx>
- Parâmetros Mínimos de Linkagem: <-lxxxx -lyyyy -Zzzzz ...>
- Autor(es):
- Versão:

2. DESCRIÇÃO GERAL

< Nesta parte do documento deve-se descrever a classe de maneira informal, dando ao usuário uma visão geral dos recursos disponíveis. **OBS: Este parágrafo deve ser apagado no documento final.**>

3. MEMBROS DA CLASSE

3.1. CONSTRUTOR(ES) E DESTRUTOR

< Nesta parte deve-se documentar o(s) construtor(es) implementado(s) para a classe, se houver(em). Em caso de não existência deve-se informar que a forma padrão é assumida. Abaixo pode ser observado o formato geral de documentação. Se houver algum comentário de interesse a respeito do destrutor, pode-se acrescentá-lo. **OBS: Este parágrafo deve ser apagado no documento final.**>

<nome da função construtora>

Propósito	<descrição resumida>
Sintaxe	*<cópia da declaração contida no arquivo header>*
Utilização	<comentários sobre o procedimento para uso da função>
Consultar	<nome de funções similares, da própria classe ou não>
Exemplo(s)	`<exemplo(s), preferencialmente simples, de utilização da função>`

<nome da função construtora herdada e não alterada>

Propósito	<descrição resumida>
Consultar	<nome da classe base>

3.2. DADOS ACESSÍVEIS

< Nesta seção deve-se documentar os dados acessíveis para utilização direta ("public"), ou via herança ("protected"). Em caso de não existência deve-se informar este fato. Abaixo pode ser observado o formato geral de documentação destes dados. **OBS: Este parágrafo deve ser apagado no documento final.**>

<nome do dado> <(public | protected)>

Descrição <descrição resumida>

Sintaxe *<cópia da declaração contida no arquivo header>*

Consultar <referência a informações pertinentes>

3.3. FUNÇÕES DE ENTRADA E SAÍDA

< Nesta parte deve-se documentar as funções destinadas à entrada e saída de dados da estrutura interna da classe. Em caso de não existência deve-se informar este fato. Abaixo pode ser observado o formato geral de documentação de uma função. **OBS: Este parágrafo deve ser apagado no documento final.**>

<nome da função> <(public | protected)>

Propósito <descrição resumida>

Sintaxe(s) *<cópia da declaração contida no arquivo header>*

Utilização <comentários sobre o procedimento para uso da função>

Retorno <valor de retorno>

Consultar <nome de funções similares, da própria classe ou não>

Exemplo(s) `<exemplo(s), preferencialmente simples, de utilização da função>`

<classe base::nome da função herdada e não alterada> <(public | protected)>

Propósito <descrição resumida>

Consultar <referência a informações pertinentes>

3.4. DEMAIS FUNÇÕES E OPERADORES

< Nesta parte deve-se documentar as funções membro (e funções amigas) que não pertencem a nenhum dos itens anteriores. Em caso de não existência deve-se informar este fato. O formato geral de documentação de função pode ser encontrado no item 3.3. **OBS: Este parágrafo deve ser apagado no documento final.**>

<4. OUTROS ITENS QUE SE FAÇAM NECESSÁRIOS, TAL COMO TRATAMENTO DE ERROS>

ANEXO I.b. - LISTA DE CLASSES C++

Código	Nome	Portabilidade			Gener.
		UNIX	DOS	Arq.	
CL-0000					
	Propósito:				
CL-0001					
	Propósito:				
CL-0002					
	Propósito:				
CL-0003					
	Propósito:				
CL-0004					
	Propósito:				
CL-0005					
	Propósito:				
CL-0006					
	Propósito:				
CL-0007					
	Propósito:				
CL-0008					
	Propósito:				
CL-0009					
	Propósito:				
CL-0010					
	Propósito:				

• • •

ANEXO II

EXEMPLO DE MANUAL DE UTILIZAÇÃO DE CLASSE C++

1. DENOMINAÇÃO DA CLASSE

- CL-0008: polynom
- Classe(s) Base: -
- Portabilidade: DOS [?] UNIX [S] (S/N/?)
- Compatibilidade de Transferência de Arquivos entre DOS e UNIX: [S] (S/N/?/-)
- Genericidade: [G] (G/X/W/O/C/...)
- Header: polynom.h
- Biblioteca: control
- Parâmetros Mínimos de Linkagem: -lcontrol -lcomplex
- Autor(es): Márcio Heidi Suguieda
- Versão: 1.1

2. DESCRIÇÃO GERAL

A classe *polynom* é a responsável pela abstração de dados do tipo polinômio de ordem positiva ou nula e com coeficientes reais. Este tipo foi desenvolvido para propiciar um conjunto de facilidades quando da manipulação de polinômios. Para tanto, foram implementados recursos de entrada e saída de dados - os operadores >> (entrar em) e << (sair de) e as funções de acesso aos dados internos (atributos) da classe - bem como recursos matemáticos - os operadores aritméticos (=, +, -, *, /, %, == e !=), as funções de teste de polinômio nulo (função *iszero*), de cálculo de raízes (função *roots*), de cálculo do valor do polinômio num dado ponto (função *polyval*), de obtenção da derivada indefinida (função *derivative*) e da integral indefinida (função *integral*)

Outra característica interessante da classe *polynom* é a flexibilidade na declaração e inicialização de dados, ou seja, existe mais de uma maneira de declarar e inicializar uma variável *polynom*. Enfim, o usuário da classe *polynom* notará que o manuseio de polinômios é bastante simples e eficiente, pois pode-se trabalhar com polinômios de forma semelhante àquela relativa aos tipos internos da linguagem C++.

Ainda convém mencionar que toda esta estrutura funciona consoante alocação dinâmica de memória, de modo que o consumo de memória disponível é otimizada.

3. MEMBROS DA CLASSE

3.1. CONSTRUTOR(ES) E DESTRUTOR

A declaração e inicialização de dados pode ser feita de diversos modos, conforme descrevem os cinco construtores a seguir:

polynom()

Propósito declarar uma variável como do tipo *polynom*

Sintaxe *polynom();*

Utilização procede-se a declaração do mesmo modo com a qual são declarados os tipos internos da linguagem C++; observa-se também que a simples declaração cria um *polynom* constante (ordem zero) nulo

Consultar demais construtores de tipo

Exemplo(s)

```
polynom aaa, bbb, ccc;
polynom pol[9];          // Array de polynom.
polynom* ptr;            // Ponteiro para um polynom.
```

polynom(double initvalue)

Propósito declarar um escalar como um polinônio de ordem zero

Sintaxe *polynom(double initvalue);*

Utilização o parâmetro *initvalue* consiste no número real a ser convertido em um polinômio de ordem zero

Consultar demais construtores de tipo

Exemplo(s)

```
polynom aa(3.2);
polynom bb(7), cc(-1.333);
```

polynom(int order, double initvalue)

Propósito declarar e inicializar um polinômio com todos coeficientes iguais

Sintaxe *polynom(int order, double initvalue);*

Utilização o parâmetro *order* (maior ou igual a zero) indica a ordem do polinômio e o parâmetro *initvalue* consiste no número a ser atribuído a todos os coeficientes do polinômio.

Consultar demais construtores de tipo

Exemplo(s)

```
polynom plx(3, 4.3);     // Polinômio de ordem 3, com
                         //  todos coeficientes = 4.3.
polynom ply(0, 1.2);     // Equivale a "polynom(1.2);".
```

polynom(int order, double* initvalues)

Propósito declarar e inicializar uma variável *polynom* a partir de um *array* (estático ou dinâmico)

Sintaxe *polynom(int order, double* initvalues);*

Utilização o parâmetro *size* (maior ou igual a zero) indica a ordem do *polynom* e, conseqüentemente, o número de dados (*order* + 1 coeficientes) apontados pelo ponteiro *initvalues*, a serem lidos. É importante observar que o índice do elemento no polinômio indica o grau do termo correspondente no polinômio

Consultar demais construtores de tipo

Exemplo(s)

```
double x[2+1];          // Ordem 2 => Tamanho 3.
// ... x[2]: coeficiente do termo de grau 2,
//     x[1]: coeficiente do termo de grau 1,
//  e  x[0]: coeficiente do termo de grau 0.
// x[2], x[1] e x[0] assumem valores quaisquer.
polynom abc(2, x);      // Polinômio de ordem 2.

double* y;
y = new double [2];     // Ordem 1 => Tamanho 2.
// ... y[1] e y[0] assumem valores quaisquer.
polynom def(1, y);      // Polinômio de ordem 1.
delete y;               // Não necessita mais de y.
```

polynom(const polynom& argm)

Propósito declarar e inicializar um *polynom* a partir de outro e garantir, quando necessária, a conversão adequada de tipos

Sintaxe *polynom(const polynom& argm);*

Utilização o parâmetro *argm* deve ser, simplesmente, outra variável do tipo *polynom*

Consultar demais construtores de tipo

Exemplo(s)

```
polynom ax(3, 1.1);
// ...
polynom by(ax);
```

IMPORTANTE:

Análises do gerenciamento interno de memória, realizado pelo compilador *Sun C++ 2.1*, mostraram que classes, as quais fazem uso de alocação dinâmica de memória, sofrem, para um dado caso, de um gerenciamento imperfeito na desalocação de áreas que perderam o escopo. Isto ocorre justamente quando os objetos (instâncias da classe) são, por sua vez, também alocados dinamicamente por um *array* de objetos.

Para solucionar este problema, recomenda-se forçar a chamada do destrutor para cada objeto do *array*, garantindo assim o gerenciamento da desalocação de memória. Os exemplos, a seguir, ilustrarão melhor o problema e a respectiva solução.

Exemplos:

```
// Exemplo 1 - O Problema!
int n;
// ... n = ...
polynom* ponteiro = new polynom [n];
// ... ponteiro[0] = polynom(...
//     ponteiro[1] = polynom(...
//     ...
//     ponteiro[n-1] = polynom(...
// ...
delete ponteiro;        // Neste caso, a desalocação
                        //  de memória não será per-
                        //  feitamente realizada.

// Exemplo 2 - A Solução!
int n;
```

```
// ... n = ...
polynom* ponteiro = new polynom [n];
// ... ponteiro[0] = polynom(...
//     ponteiro[1] = polynom(...
//     ...
//     ponteiro[n-1] = polynom(...
// ...
// Força o destrutor para cada ponteiro[i].
polynom* destroi;
for (int i = 0; i < n; i++) {
  destroi = &ponteiro[i];
  destroi->polynom::~polynom();
}
delete ponteiro;        // Agora tudo OK!
ponteiro = NULL;        // Robustez na programação.
```

3.2. DADOS ACESSÍVEIS

Abaixo estão as variáveis (protegidas) da classe, que podem eventualmente serem usadas por classes herdeiras desta.

degree	(protected)
Descrição	ordem do polinômio
Sintaxe	*int degree;*
Consultar	-

coeff	(protected)
Descrição	ponteiro para os coeficientes
Sintaxe	*double* coeff;*
Consultar	-

3.3. FUNÇÕES DE ENTRADA E SAÍDA

Para a classe *polynom*, estão disponíveis os operadores >> (entrar em) e << (sair de), responsáveis pela entrada e saída de dados, respectivamente. Note que estes operadores foram sobrecarregados para entrada ou saída de dados, tanto via *console* como via arquivo ASCII. Basicamente, a leitura ou escrita de um *polynom* é feita com base no **formato**

$$(a_n \; a_{n-1} \; \ldots \; a_0)$$

onde $a_n \; a_{n-1} \; \ldots \; a_0$ são os coeficientes reais do polinômio em ordem **decrescente** de grau, ou seja, a_n corresponde ao coeficiente do termo de maior grau e a_0 corresponde ao coeficiente do termo de grau zero. Note também que os parênteses são partes integrantes do formato de um *polynom*.

operator >> (public)

Propósito propiciar a entrada de dados via *console*

Sintaxe(s) *friend istream& operator>> (istream& in, polynom& argm);*

Utilização utiliza-se o operador >> para o tipo *polynom* consoante a forma descrita pelas classes *stream*

Retorno retorna condição de erro, segundo o funcionamento das classes *stream*

Consultar classes *stream*, *operator* <<

Exemplo(s)

```
polynom aa, bb;
// ...
// Formato de um polynom: (a b c d ...) onde
//  a, b, c, d, ... são os coeficientes do
//  polinômio em ordem decrescente de grau.
cin >> aa;                    // Sem tratamento de
erro.
if (!(cin >> bb)) { // Tratamento de erro! }
```

operator >> (public)

Propósito propiciar a entrada de dados via arquivo ASCII

Sintaxe(s) *friend ifstream& operator>> (ifstream& fin, polynom& argm);*

Utilização utiliza-se o operador >> para o tipo *polynom* consoante a forma descrita pelas classes *stream*. E a propósito, o usuário está livre para criar os seus arquivos no formato que achar mais conveniente

Retorno retorna condição de erro, segundo o funcionamento das classes *stream*

Consultar classes *stream*, *operator* <<

Exemplo(s)

```
polynom xyz;
ifstream infile("arquivo.tst");
if (!infile) {
   cerr << "Impossivel abrir o arquivo!" << endl;
   exit(1);
}
// Formato de um polynom: (a b c d ...) onde
//  a, b, c, d, ... são os coeficientes do
//  polinômio em ordem decrescente de grau.
if (!(infile >> xyz)) {
   cerr << "Problemas com o arquivo!" << endl;
   exit(1);
}
infile.close();
```

operator << (public)

Propósito propiciar a saída de dados via *console*

Sintaxe(s) *friend ostream& operator<< (ostream& out, const polynom& argm);*

Utilização utiliza-se o operador << para o tipo *polynom* consoante a forma descrita pelas classes *stream*

Retorno retorna condição de erro, segundo o funcionamento das classes *stream*

Consultar classes *stream*, *operator >>*

Exemplo(s)

```
polynom abc;
// ...
cout << "Polinomio: " << abc << endl;
          // A saída de dados segue o formato
          //  descrito no início desta seção.
```

operator << (public)

Propósito propiciar a saída de dados via arquivo ASCII

Sintaxe(s) *friend ofstream& operator<< (ifstream& fout, const polynom& argm);*

Utilização utiliza-se o operador << para o tipo *polynom* consoante a forma descrita pelas classes *stream*. E a propósito, o usuário está livre para criar os seus arquivos no formato que achar mais conveniente.

Retorno retorna condição de erro, segundo o funcionamento das classes *stream*

Consultar classes *stream*, *operator >>*

Exemplo(s)

```
polynom xyz;
ofstream outfile("arquivo.tst");
if (!outfile) {
   cerr << "Impossivel abrir o arquivo!" << endl;
   exit(1);
}
if (!(outfile << xyz)) {
   cerr << "Problemas com o arquivo!" << endl;
   exit(1);
}
          // A saída de dados segue o formato
          //  descrito no início desta seção.
outfile.close();
```

3.3.1. FUNÇÕES DE ACESSO AOS DADOS (ATRIBUTOS) DA CLASSE

A seguir, estão descritos as funções que permitem, de alguma forma, o acesso aos dados internos (atributos) da classe:

order (public)

Propósito fornecer a ordem do *polynom*

Sintaxe(s) *int order() const;*

Utilização ver exemplo

Retorno retorna um inteiro correspondente a ordem do polinômio

Consultar -

Exemplo(s)

```
polynom pol;
// ...
int ordem = pol.size();
```

getval (public)

Propósito permitir a leitura de coeficiente do polinômio

Sintaxe(s) *double getval(int index) const;*

Utilização o parâmetro *index* (maior ou igual a zero) corresponde ao grau do termo cujo coeficiente é desejado

Retorno se o coeficiente existe retorna o valor; caso contrário, retorna zero (os coeficientes não existentes de um polinômio são naturalmente zeros)

Consultar *setval*

Exemplo(s)

```
polynom xyz;
// ...
double coef6, coef0;
coef6 = xyz.getval(6);  // Coeficiente do termo
                        //  de grau seis.
coef0 = xyz.getval(0);  // Coeficiente do termo
                        //  de grau zero.
```

setval (public)

Propósito permitir a escrita de coeficiente em qualquer termo do polinômio

Sintaxe(s) *void setval(int index, double value);*

Utilização o parâmetro *index* (maior ou igual a zero) corresponde ao grau do termo cujo coeficiente será atribuído o valor *value*

Retorno -

Consultar *getval*

Exemplo(s)

```
polynom abc;
// ...
abc.setval(0, -2.9);    // Coeficiente de grau zero
                        //  passa a ser -2.9.
abc.setval(3, -3.1);    // Coeficiente de grau três
                        //  passa a ser -3.1.
```

3.4. DEMAIS FUNÇÕES E OPERADORES

Para esta classe, ainda foram sobrecarregados operadores de modo a garantir uma aritmética básica entre polinômios e também criadas algumas funções para manipulação algébrica dos mesmos.

operator = (public)

Propósito realizar a atribuição de polinômios

Sintaxe(s) *polynom operator= (const polynom& rval);*

Utilização atribuição simples de polinômios, onde estes não necessitam ser de mesma ordem

Retorno cópia do polinômio resultante da atribuição

Consultar demais operadores aritméticos

Exemplo(s)

```
polynom a(3.3), b(7, 2.3), c;
// ...
c = a;
a = b;
```

operator + (public)

Propósito realizar a adição de polinômios

Sintaxe(s) *friend polynom operator+ (const polynom& lval, const polynom& rval);*

Utilização pode-se proceder tanto a soma de polinômios como a soma de polinômio com escalar e vice-versa

Retorno cópia do polinômio resultante da adição

Consultar demais operadores aritméticos

Exemplo(s)

```
polynom x(1.1), y(3, -2.2), z;
// ...
z = x + y;
z = x + 1;
z = 1.3 + y + x;
```

operator - (public)

Propósito realizar a subtração de polinômios ou o unário menos

Sintaxe(s) *friend polynom operator- (const polynom& lval, const polynom& rval);*

polynom operator- () const;

Utilização pode-se proceder tanto a subtração de polinômios como a subtração de polinômio por escalar (e vice-versa) ou ainda o unário menos

Retorno cópia do polinômio resultante da subtração

Consultar demais operadores aritméticos

Exemplo(s)

```
polynom x(7.3), y(7, -1.2), z;
// ...
z = x - y;
z = x - 1;
z = 2.3 - x - y;
z = -z;                    // Unário menos.
```

operator * (public)

Propósito realizar a multiplicação de polinômios

Sintaxe(s) *friend polynom operator* (const polynom& lval, const polynom& rval);*

Utilização pode-se proceder tanto o produto de polinômios como o produto de polinômio por escalar e vice-versa

Retorno cópia do polinômio resultante da multiplicação

Consultar demais operadores aritméticos

Exemplo(s)

```
polynom w(3, 1.1), z;
// ...
z = 7.2 * w * w;
z = -z * w;
z = w * 0.5;
```

operator / (public)

Propósito realizar a divisão de polinômios

Sintaxe(s) *friend polynom operator/ (const polynom& lval, const polynom& rval);*

Utilização pode-se proceder tanto a divisão de polinômios como a divisão de polinômio por escalar e vice-versa. Observe que a divisão de polinômios não é necessariamente exata e, conseqüentemente, pode apresentar resto

Retorno cópia do polinômio resultante da divisão

Consultar demais operadores aritméticos (em especial o *operator%*)

Exemplo(s)

```
polynom w(5, -2.5), y(3.2, 5), quociente;
// ...
quociente = w / y;
quociente = y / w;
quociente = w / 5.2;
quociente = 5.2 / y;
```

operator % (public)

Propósito realizar a operação resto (ou módulo) da divisão de polinômios

Sintaxe(s) *friend polynom operator% (const polynom& lval, const polynom& rval);*

Utilização pode-se obter tanto o resto da divisão de polinômios como o resto da divisão de polinômio por escalar e vice-versa. Observe que a divisão de polinômios não é necessariamente exata e, conseqüentemente, pode apresentar resto

Retorno cópia do polinômio resultante da operação resto da divisão

Consultar demais operadores aritméticos (em especial o *operator/*)

Exemplo(s)

```
polynom w(5, -2.5), y(3.2, 5), resto;
// ...
resto = w % y;
resto = y % w;
resto = w % 5.2;
resto = 5.2 % y;
```

operator == (public)

Propósito propiciar a comparação de igualdade de polinômios

Sintaxe(s) *friend int operator== (const polynom& lval, const polynom& rval);*

Utilização ver exemplo

Retorno zero se for falsa e a unidade se for verdadeira

Consultar *iszero*, demais operadores aritméticos

Exemplo(s)
```
polynom x(2, 1.1), y(2, -2.2);
// ...
if (x == y) { // ... }
```

operator != (public)

Propósito propiciar a comparação de desigualdade de polinômios

Sintaxe(s) *friend int operator!= (const polynom& lval, const polynom& rval);*

Utilização ver exemplo

Retorno zero se for falsa e a unidade se for verdadeira

Consultar *iszero*, demais operadores aritméticos

Exemplo(s)
```
polynom x(2, -2.1), y(2, -1.2);
// ...
if (x != y) { // ... }
```

iszero (public)

Propósito testar se o polinômio é uma constante nula

Sintaxe(s) *int iszero() const;*

Utilização ver exemplo

Retorno zero se for falso (polinômio não nulo) e a unidade se for verdadeiro (polinômio constante nulo)

Consultar *operator==*, *operator!=*

Exemplo(s)
```
polynom z;
// ...
if (z.iszero()) { // ... polinômio z é zero!}
```

polyval (public)

Propósito calcular o valor (complexo) do polinômio num dado ponto (complexo)

Sintaxe(s) *complex polyval(complex value) const;*

Utilização ver exemplo

Retorno um *complex* correspondente ao valor do polinômio no ponto

Consultar *roots*, classe *complex*

Exemplo(s)
```
polynom pol;
// ...
complex valor = pol.polyval(complex(-2.2, 4.5));
```

roots (public)

Propósito calcular as raízes (complexas) do polinômio

Sintaxe(s) *cvector roots() const;*

Utilização declarar uma variável do tipo *cvector* e atribuir o valor de retorno a esta variável; desse modo, as raízes serão alocadas num vetor de números complexos (tipo *cvector*)

Retorno um *cvector* contendo as raízes (complexas) em ordem decrescente de módulo

Consultar *polyval*, classe *complex*, classe *cvector*

Exemplo(s)

```
polynom xxx;
// ... xxx assume um polinômio qualquer.
cvector raizes = xxx.roots();
// ... raizes[i], i = 0, 1, ..., xxx.order(),
//     serão números complexos denotando as
//     raízes do polinômio em questão.
complex uma_raiz = raizes[0];
// ...
```

derivative (public)

Propósito obter a derivada indefinida do polinômio

Sintaxe(s) *polynom derivative() const;*

Utilização ver exemplo

Retorno um *polynom* contendo a derivada indefinida do polinomio em questão

Consultar *integral*

Exemplo(s)

```
polynom polin(3, 2), deriv;
// ...
deriv = polin.deriv();
```

integral (public)

Propósito obter a integral indefinida do polinômio

Sintaxe(s) *polynom integral(double constant = 0.) const;*

Utilização como a integral indefinida de um polinômio pode apresentar, a princípio, qualquer termo de ordem zero (termo constante), então é opcional ao usuário fornecer o parâmetro *constant* a ser assumido como o coeficiente de grau zero do polinômio resultante da integração indefinida

Retorno um *polynom* contendo a integral indefinida do polinomio em questão

Consultar *integral*

Exemplo(s)

```
polynom polin(3, 2), integ;
// ...
integ = polin.integral();      // Coef. grau zero = 0.
integ = polin.integral(-2);    // Coef. grau zero = -2.
```

4. TRATAMENTO DE ERROS

A classe *polynom* possui tratamento de erros para auxílio na depuração de programas. Sucintamente, as mensagens de erro ou de simples aviso são as seguintes:

- *Negative order!* - definido um polinômio com ordem negativa e, portanto, não válida para esta classe.

- *Index out of range for setval()!* - uso de índice negativo na função *setval*.

- *Division by null polynomial (zero)!* - tentativa errônea de divisão do polinômio por um polinômio ou escalar nulo.

- *Invalid remainder (division by zero)!* - tentativa errônea de obtenção do resto da divisão do polinômio por um polinômio ou escalar nulo.

- *Roots for constant are not defined!* - tentativa errônea de obtenção de raízes de um polinômio de grau zero, ou seja, uma constante.

- *Roots convergence problems! Trying again ...* - o cálculo das raízes não convergiu em 30 iterações, porém uma nova tentativa de obtenção das raízes estará em andamento.

O tratamento de erro para os operadores >> e << também é feito, onde o princípio de funcionamento é o mesmo adotado pelas classes *stream*.

ANEXO III

EXEMPLO DE *MAN PAGES* DE CLASSE C++

```
POLYNOM(3C++)          Classe C++ : CL-0008          POLYNOM(3C++)

NOME
polynom - abstracao do tipo polinomio de ordem positiva ou nula
e com coeficientes reais

SINOPSE
#include <polynom.h>

class polynom {
     public:
          polynom();
          polynom(double initvalue);
          polynom(int order, double initvalue);
          polynom(int order, double* initvalues);
          polynom(const polynom& argm);
          virtual ~polynom();

          friend istream& operator>> (istream& in,
                                              polynom& argm);
          friend ostream& operator<< (ostream& out,
                                         const polynom& argm);
          friend ifstream& operator>> (ifstream& fin,
                                              polynom& argm);
          friend ofstream& operator<< (ofstream& fout,
                                         const polynom& argm);
          int    order() const;

          double getval(int index) const;
          void   setval(int index, double value);

          polynom operator= (const polynom& rval);

          friend polynom operator+ (const polynom& lval,
                                         const polynom& rval);

          friend polynom operator- (const polynom& lval,
                                         const polynom& rval);
          polynom operator- () const;

          friend polynom operator* (const polynom& lval,
                                         const polynom& rval);

          friend polynom operator/ (const polynom& lval,
                                         const polynom& rval);
          friend polynom operator% (const polynom& lval,
                                         const polynom& rval);

          friend int operator== (const polynom& lval,
                                         const polynom& rval);
          friend int operator!= (const polynom& lval,
                                         const polynom& rval);

LCMI              Last change: 15 Agosto 1993                1
```

```
POLYNOM(3C++)          Classe C++ : CL-0008          POLYNOM(3C++)

          int      iszero() const;

          cvector roots() const;

          complex polyval(complex value) const;

          polynom derivative() const;
          polynom integral(double constant = 0.) const;

     protected:
          int          degree;
          double*      coeff;

};

DESCRICAO
As funcoes desta classe podem ser encontradas em
"libcontrol.a", devendo esta biblioteca ser "linkada" com os
demais modulos atraves da opcao -lcontrol. Alem da opcao
-lcontrol, deve-se tambem acrescentar a opcao -lcomplex,
garantindo a definicao dos numeros complexos e,
consequentemente, das funcoes que fazem uso de numeros
complexos.
As declaracoes desta classe sao encontradas no arquivo de
inclusao <polynom.h>. Em funcao destas consideracoes, torna-se
necessario utilizar as opcoes -I<path>/include na compilacao e
-L<path>/lib na linkagem, onde <path> e' o caminho para a area
onde estao armazenadas as classes CL-XXXX.
O tipo "polynom" foi criado para atender as necessidades de
polinomios, baseados em alocacao dinamica de memoria, que
exijam recursos de entrada e saida de dados, assim como de uma
aritmetica basica e sob uma estrutura flexivel, eficiente e
robusta.

- CONSTRUTORES DE TIPO
Dentre os recursos propiciados, estao cinco contrutores de
tipo: o primeiro apenas cria uma variavel como do tipo
"polynom", o segundo converte um "double" em um "polynom", o
terceiro cria um "polynom" com todos os coeficientes iguais, o
quarto o faz a partir de um "array" de "doubles", onde o indice
de cada elemento (coeficiente) corresponde ao grau do termo
associado, e por fim, o quinto e ultimo cria um "polynom" a
partir de outro ja' existente.

Exemplos:

  polynom pl1;          // Apenas cria um polynom.

  polynom pl2(3);       // Converte o 3 num polynom.
                        //  (polinomio de grau zero)

LCMI              Last change: 15 Agosto 1993              2
```

```
POLYNOM(3C++)          Classe C++ : CL-0008          POLYNOM(3C++)

   polynom pl3(5, 1.2);   // Cria um polynom de grau 5,
                          //  com coefs. iguais a 4.2.

   double x[2+1];         // Grau n ==>> n + 1 coeficientes.
   x[0] = 0;              // Coeficiente do termo de grau 0.
   x[1] = 1;              // Coeficiente do termo de grau 1.
   x[2] = 2;              // Coeficiente do termo de grau 2.
   polynom pl4(2, x);     // Cria e inicializa um polynom, a
                          //  partir de um array estatico.

   double* y = new double [1+1];
   y[0] = 0.12;           // Coeficiente do termo de grau 0.
   y[1] = 0.34;           // Coeficiente do termo de grau 1.
   polynom pl5(2, y);     // Cria e inicializa um polynom, a
   delete y;              //  partir de um array dinamico.

   polynom pl6(pl5);      // Cria um polynom a partir de
                          //  outro ja existente.

- ENTRADA E SAIDA DE DADOS
Para a entrada e saida de dados, encontram-se disponiveis os
operadores >> e <<, que funcionam consoante o padrao das
classes "stream". Estes operadores podem ser utilizados tanto
para leitura/escrita por meio do "console" como por meio de
arquivos. Basicamente, um "polynom" deve ser lido/escrito com
base no formato (a b c ... z) onde a, b, c, ..., z sao os
coeficientes reais (tipo "double") do polinomio em ordem
DECRESCENTE de grau, ou seja, o coeficiente "a" e' relativo ao
termo de maior grau e o "z" e' relativo ao termo de grau zero.
Observe também que os parenteses sao parte integrante do
formato dado acima.

Exemplos:

   polynom abc;
   cout << "Entre com o polinomio "
           "(ordem DECRESCENTE de grau): " << flush;
   cin >> abc;
   cout << "O polinomio lido foi: " << abc << endl;
   polynom def;
   // ...
   ofstream tstout("teste.xxx");
   cout << "Escrevendo arquivo: teste.xxx" << endl;
   tstout << def;
   tstout.close();
   ifstream tstin("teste.xxx");
   cout << "Lendo arquivo: teste.xxx" << endl;
   tstin >> def;
   cout << "O polinomio lido foi: " << def << endl;
   tstin.close();

LCMI              Last change: 15 Agosto 1993              3
```

```
POLYNOM(3C++)          Classe C++ : CL-0008          POLYNOM(3C++)

- OPERADORES E FUNCOES DE ACESSO
O acesso a um elemento do polinomio pode ser feito pelas
funcoes getval() e setval() responsaveis, respectivamente, pela
leitura e escrita de coeficientes no polinomio. A unica
restricao esta' no uso indevido de indices negativos, quando da
escrita de um coeficiente. Por outro lado, se for necessario
acessar a ordem do polinomio basta fazer uso da funcao
order().

- OPERADORES ARITMETICOS
Para esta classe, estao presentes o operador de atribuicao =,
os operadores aritmeticos +, -, *, / e %, e os operadores de
comparacao == e !=. Dentre estes operadores, convem salientar
que o operador % (conhecido como modulo ou resto) foi
implementado para retornar o resto da divisao de polinomios,
pois uma divisao de polinomios nao necessariamente retorno
resto nulo.

Exemplos:

   polynom A, B;
   A.setval(0, 0.);
   A.setval(1, -2.);
   A.setval(2, 5);
   // ...
   B = -A;
   B = A + B;
   A = A - B * 2;   // Vale a precedencia de sinal.
   A = A * B + 2.5 * A - 3;
   B = B / A;
   B = B % A;       // Resto da divisao.
   B = 7.5 / A;
   B = 2 - 2.2 % A;
   // ...
   if (A != B) { //... }

- FUNCOES MATEMATICAS
Alem dos operadores aritmeticos, encontram-se disponiveis as
seguintes funcoes matematicas (ou logicas): iszero() verifica
se o polinomio e' uma constante nula, polyval() calcula o valor
(complexo) do polinomio num dado ponto (complexo), roots() -
calcula as raizes (complexas) do polinomio, derivative() -
calcula a derivada indefinida do polinomio e integral() -
calcula a integral indefinida do polinomio, com a opcao do
usuario fornecer a constante (coeficiente de grau zero
opcionalmente nulo) do polinomio. Ainda, convem ressaltar que
no caso da funcao roots(), o valor de retorno sera' um
"cvector", ou seja, um vetor de numeros complexos contendo as
raizes do polinomio em ordem decrescente de modulo (ver as "man
pages" da classe "cvector" CL-0005).

LCMI              Last change: 15 Agosto 1993              4
```

```
POLYNOM(3C++)          Classe C++ : CL-0008          POLYNOM(3C++)

- TRATAMENTO DE ERRO
A classe "polynom" possui tratamento de erros para auxilio na
depuracao de programas. Sucintamente, as mensagens de erro ou
de simples aviso de problemas sao as seguintes:

Negative order! - definido um polinomio com ordem negativa e,
portanto, nao valida para esta classe.

Index out of range for setval()! - uso de indice negativo na
funcao setval.

Division by null polynomial (zero)! - tentativa erronea de
divisao do polinomio por um polinomio ou escalar nulo.

Invalid remainder (division by zero)! - tentativa erronea de
obtencao do resto da divisao do polinomio por um polinomio ou
escalar nulo.

Roots for constant are not defined! - tentativa erronea de
obtencao de raizes de um polinomio de grau zero, ou seja, uma
constante.

Roots convergence problems! Trying again... - o calculo das
raizes nao convergiu em 30 iteracoes, porem uma nova tentativa
de obtencao das raizes estara' em andamento.

O tratamento de erro para os operadores >> e << tambem e'
feito, onde o principio de funcionamento e' o mesmo adotado
pelas classes "stream".

CONSULTAR
     IOS.INTRO(3C++), CPLX.INTRO(3C++), cvector(3C++)

LCMI              Last change: 15 Agosto 1993              5
```